Spis treści

JavaScript 1.1
Wprowadzenia – o zawartości podręcznika 1.2
- Licencja 1.2.1
- Terminologia 1.2.2

JavaScript jako element strony internetowej 1.3
- jQuery 1.3.1
- Struktury, JSON 1.3.2
- Ajax 1.3.3
- API 1.3.4

Podstawy Java Script 1.4
- Javascript jako kalkulator 1.4.1
- Podstawowe operacje na napisach 1.4.2
- Instrukcje warunkowe i wyrażenia logiczne 1.4.3
- Kolejność działań 1.4.4
- Typy danych 1.4.5
- Funkcje i parametry 1.4.6
- Instrukcje i algorytmy 1.4.7
- Callback 1.4.8

Rozwój języka 1.5
- Funkcje, klasy i obiekty 1.5.1
- Dialekty 1.5.2
- ES2015 1.5.3
- TypeScript 1.5.4

Bloki i zakres widoczności zmiennych 1.6

Notacja kropkowa 1.7

Deklaratywny styl programowania (AngularJS) 1.8
- React - wprowadzenie 1.8.1
- Angular - wprowadzenie 1.8.2

Programowanie funkcyjne 1.9

Redux 1.9.1
Redux i Anguler 1.9.1.1
TDD 1.10
Co dalej? 1.11

UCZYMY SIĘ PROGRAMOWAĆ W JAVASCRIPT

Otwarty podręcznik programowania

(C) Jerzy Wawro

BRUDNOPIS

Podręcznik jest skończony w około 80%

Wprowadzenia – o zawartości podręcznika

Tworząc ten podręcznik zamierzałem wzorować się na opracowaniu “Uczymy się programować w Pythonie”: koncentrujemy się na nauce programowania, a sam język programowania poznajemy niejako przy okazji. Szybko jednak okazało się, że przejście od Pythona do Javascript ma szersze konsekwencje, niż tylko konieczność napisania przykładów na nowo. Musimy zwracać niestety więcej uwagi na szczegóły techniczne – nie wiązane wprost z logiką programu. Wynika to wprost z przeznaczenia i strategii rozwoju języka. Javascript działa w środowisku przeglądarki internetowej, które ma swoje ograniczenia. Nie przewidziano na przykład prostych mechanizmów wprowadzania danych z konsoli [zob. https://www.w3schools.com/jsref/met_win_prompt.asp, https://nodejs.org/api/readline.html]. Twórcy języka muszą także dbać o wsteczną zgodność, aby nie spowodować tego, że po wejściu w życie nowego standardu większość stron internetowych przestanie poprawnie się wyświetlać. W roku 2015 wszedł w życie standard języka ES6, który wprowadził wiele nowoczesnych rozwiązań, ułatwiających programowanie. Jednak ten standard nadal nie jest zaimplementowany wprost w przeglądarkach, a programy zgodnie z nim napisane są automatycznie tłumaczone tak – by były zgodne z możliwościami przeglądarek.

Wszystkie te ograniczenia wpłynęły na zmianę koncepcji podręcznika. Nie jest to elementarz dla osób uczących się programować (do tego zadania język Python jest jednak lepszy), ale elementarny podręcznik dla osób chcących nauczyć się tworzenia aplikacji mobilnych i internetowych. Nie uczymy się zatem ogólnych zasad programowania, ale zasad tworzenia programów w konkretnym środowisku.

Podręcznik składa się z następujących części:

1. Javascript jako element strony internetowej. Zamiast abstrakcyjnych przykładów mamy w tym rozdziale krótki opis struktury strony internetowej. Poznajemy przy okazji podstawowe terminy dotyczące zastosowań Javascript.

2. W drugiej części podręcznika opisano podstawowe koncepcje: wyrażenia, zmienne, typy danych, funkcje i instrukcje. Są to elementy występujące nie tylko we wszystkich standardach Javascript, ale w większości języków programowania.

3. Kolejny rozdział - „Rozwój języka" opisuje przeobrażenia jakie przeszedł Javascript. W podrozdziale "Funkcje i obiekty” - przedstawiono specyficzną dla Javascript koncepcję użycia funkcji w charakterze obiektów. To są zagadnienia trudne, ale nie sposób o nich nie wspomnieć. Choćby po to, by uzasadnić konieczność wprowadzenia nowego standardu (ES6). Osoby dopiero zaczynające przygodę z programowaniem mogą ten rozdział pominąć.

Wydany w 2015 roku standard ES6 (ES2015) sprawił, że Javascript stał się nowoczesnym językiem programowania, w którym można równie dobrze tworzyć duże programy jak i nauczać elementów programowania. Dalej opisano Typescript, który jest dialektem Javascript.

4. Dopiero reszta podręcznika jest w pełni wierna koncepcji nauczania przez proste przykłady (przykłady używane wcześniej służyły jedynie ilustracji omawianych zagadnień). Przykłady zostały stworzone w Javascript ES6 i dwóch ważnych „dialektach” tego języka: React i Typescript. Przykłady ilustrują podstawowe idee programowania aplikacji webowych.

Każda z opisanych "technologii" jest przedmiotem wielu dokumentacji i podręczników. Niniejszy podręcznik nie ma na celu ani ich zastępować, ani uzupełniać. To propozycja zupełnie innej ścieżki wchodzenia w świat Javascript: poprzez uchwycenie najważniejszych idei i ich zastosowanie w praktyce.

W nauce programowania najważniejszy jest jednak ten moment, kiedy nasz program wykona coś użytecznego. Dlatego w ostatniej części podręcznika posługujemy się przykładami z jednej strony prostymi a z drugiej kompletnymi.

Licencja

Podręcznik jest udostępniany na licencji *Creative Commons* – zwanej w poniższym objaśnieniu warunków wykorzystania „Licencją”. Podręcznik określa się jako „Dzieło”. Użytkownik (czytelnik) korzystając z podręcznika automatycznie akceptuje warunki Licencji – stając się w ten sposób Licencjobiorcą.

Zgodnie z Licencją:

Licencjobiorca ma obowiązek zachować w stanie nienaruszonym wszelkie oznaczenia związane z prawno-autorską ochroną Dzieła oraz zapewnić, stosownie do możliwości używanego nośnika lub środka przekazu oznaczenie:

- imienia i nazwiska (lub pseudonimu, odpowiednio) Twórców, jeżeli zostały one dołączone do Utworu, oraz (lub) nazwę innych podmiotów jeżeli Twórca oraz (lub) Licencjodawca wskażą w oznaczeniach związanych z prawno-autorską ochroną Utworu, regulaminach lub w inny rozsądny sposób takie inne podmioty (np. sponsora, wydawcę, czasopismo) celem ich wymienienia ("Osoby Wskazane");
- tytułu Utworu, jeżeli został dołączony do Utworu;
- w rozsądnym zakresie URI, o ile istnieje, który Licencjodawca wskazał jako związany z Utworem, chyba że taki URI nie odnosi się do oznaczenia związanego z prawno-autorską ochroną Utworu lub do informacji o zasadach licencjonowania Utworu; oraz
- w przypadku Utworu Zależnego, oznaczenie wskazujące na wykorzystanie Utworu w Utworze Zależnym (np. "francuskie tłumaczenie Utworu Twórcy," lub "scenariusz na podstawie Utworu Twórcy").

Licencjobiorca może więc zgodnie z Licencją wykorzystywać i rozpowszechniać Dzieło pod warunkiem dołączenia kopii Licencji lub wskazania adresu strony internetowej (URI), pod którym znajduje się tekst Licencji do każdego egzemplarza Dzieła.

Licencjobiorca nie może oferować ani narzucać żadnych warunków lub ograniczeń nie uwzględnionych w Licencji. Nie może też udzielać sublicencji.

Niniejszy opis zawiera jedynie omówienie najważniejszych zapisów Licencji. Pełna jej treść jest dostępna na stronie: *Creative Commons Attribution-ShareAlike 4.0 International License* Polskie tłumaczenie (do wersji 3.0): *https://creativecommons.org/licenses/by-sa/3.0/pl/legalcode*.

Prawa uzytkowania i powielania

> Licencjobiorca ma prawo do wyświetlania tej książki, jej kopiowania i dystrybucji, tak długo, jak podaje oryginalnych autorów. Może modyfikować tekst książki, tworząc prace pochodne, o ile opisze różnice i opublikuje pracę na takiej samej licencji.

Kod programów

> ## Licencjonowanie kodu
>
> Cały kod / skrypty znajdujące się w tej książce jest udostępniany na licencji BSD, chyba że zaznaczono inaczej.

Terminologia

Podręcznik jest tworzony w języku polskim. Jednak w niektórych przykładach zdecydowano się stosować identyfikatory zmiennych bazujące na języku angielskim. Te kilkadziesiąt słów „współczesnej łaciny” łatwo opanować nawet komuś, kto nie zna języka angielskiego. Zastosowanie takiego rozwiązania ułatwi natomiast integracje z międzynarodowym środowiskiem programistów Javascript.

Z angielskiego pochodzą także tak zwane ‘słowa kluczowe’ - używane do zaznaczenia elementów struktury programu. Nie należy używać tych słów jako identyfikatorów. Pomimo, że Javascript rozróżnia duże i mal litery i teoretycznie identyfikator **For** nie będzie się mylił ze słowem kluczowym **for** – mogłoby to prowadzić do dodatkowych błędów w razie pomyłki.

Spis wszystkich słów kluczowych najdziesz na stronie:
https://www.w3schools.com/js/js_reserved.asp

JavaScript jako element strony internetowej

Struktura strony internetowej

W projekcie każdej strony internetowej można wyróżnić trzy główne aspekty (warstwy): treść (content), wygląd (styl) oraz sposób zachowania (dynamika).

Treść opisujemy przy pomocy języka HTML a wygląd za pomocą CSS. Za aspekt dynamiczny odpowiada głównie JavaScript.

Mówiąc o dynamicznych zmianach i JavaScrpt trzeba pamiętać, żemowa jest tu o zmianach (ruchu) w aktualnie oglądanej stronie (po jej wczytaniu). Nie należy tego utożsamiać z podziałem stron na statyczne i dynamicze. Strony dynamiczne to takie, które są tworzone na serwerze z danych - najczęściej pamiętanych w bazie danych. Takim tworzeniem stron zajmują się na przykład systemy CMS, których popularnym przedstawicielem jest Wordpress. Tym nie będziemy się zajmowali (tak zwany backend). Wspominamy o tym wyłącznie dlatego, aby pokazać typowy kontekst użycia JavaScript.

> Dygresja: Strony statyczne to takie, które nie są tworzone przez programy w trakcie wczytywania, ale ich treść została wcześniej w pełni przygotowana i zapisana w plikach HTML i CSS. Takie strony także mogą siż zmieniać – na przykład przy zmianie wielkości ekranu. Strony dynamiczne to takie których zachowanie lub wygląd definiuje program. Tu sprawa nieco się komplikuje, bo program może być wykonywany przez przeglądarkę internetową (frontend) lub serwer www (backend). W niniejszym podręczniku mowa właśnie o tym, co dzieje się po stronie frontendu -gdzie używany jest JavaScript. Czy stronę zawierającą bogatą aplikację w JavaScript nazwać statyczną czy dynamiczną? To nie ma większego znaczenia (często przyjmuje się po prostu, że dynamiczne = z treścią pamiętaną w bazie danych).

Krótki wstęp do HTML

Pierwotnie wzorem dla stron internetowych były teksty tworzone w edytorach (takich jak Word), które dzielone są na paragrafy. Do zbudowania strony wystarczały**znaczniki (ang. “tag”)**:

- podział na paragrafy (p);
- załamanie wiersza (br);
- podkreślenia, kursywa, wytłuszczenia (u,i,b);

- nagłówki (h1,h2,...);
- linki (a);
- wyliczenia i numerowania (ul, ol, li);
- wstawienie obrazka (img);
- tablice (table, tr,th,td);
- formularze (form, input, button);

Każdy znacznik ma dwa warianty – określające początek I koniec tekstu. Na przykładzie znacznika p:

`<p>` - początek paragrafu

`</p>` - koniec paragrafu

Czyli używamy znaków mniejszy (<) I większy (>) aby odróżnić znacznik od reszty tekstu. Aby uniknąć pomyłek – w tekście (poza znacznikami) zamiast znaku mniejszości stosujemy zapis: `<.`

Przykład paragrafu w HTML:

```
<p><b>2/5 jest &lt; niż 2/3</b></p>
```

Z czasem strony internetowe zaczęły przypominać publikacje drukowane: podzielone na ramki.

Do określenia ramek stosuje się znacznik **div**.

```
<div>
<div>
 ramka 1
</div>
<div>
 ramka 2
</div>
</div>
```

Domyślnie ramki ustawiają się jedna pod drugą.

Fragmenty tekstów w obrębie ramek można wyróżniać znacznikiem `span.` Na przykład:

```
To jest <span> wyróżniony tekst</span>
```

Sposób wyświetlania ramek i tekstów (span) ustawia się przy pomocy styli - czyli własności znaczników o nazwie "style"

Przykład:

```
<div>To jest <span style="color:red">czerwony tekst</span></div>
```

Na tym przykładzie widać, że **własność**znaczników to dodatkowy opis między znakami <> - po identyfikatorze znacznika i oddzielony od niego spacją.

Przykłady własności niektórych znaczników:

- `<a href="adres internetowy strony">`
- `<img src="adres internetowy zdjęcia" alt="opis zdjęcia>`
- `<form action="adres internetowy dokąd wysyłamy dane z formularza" method="sposób wysyłania">`

Oczywiście w miejsce opisów jak powyżej należy wpisać konkretne informacje.

W przypadku formularza (form) pokazano jak podzielić opis znacznika na dwa wiersze. Sposób wysyłania w tym znaczniku może być określony słowem "GET" lub "POST" (zalecane).

Poza wspomnianymi powyżej, najczęściej stosujemy własności identyfikujące znacznik. Mamy przy tym dwa rodzaje własności identyfikatorów :

- identyfikator konkretnego znacznika: id
- identyfikator klasy: class

Przykład:

```
<div id="ramka1">
To jest <span class="kolorowe_teksty" style="color:red">czerwony tekst</span>
</div>
```

Poza własnościami styli i identyfikatorów niezbędne są:

- własność **href** - dla znacznika `<a>` - określa link (URL) docelowy
- własność **src** dla znacznika `img` - określa adres grafiki
- własności związane z formularzami (`action, method, type, value, name`)
- własności związane z tabelami (na przykład łączenie komórek).

Dodatkowo w dalszej części podręcznika pojawią się własności opisujące zdarzenia związane z funkcjonowaniem strony. Na przykład własność „**onclick**” opisuje co stanie się, gdy w element opisany znacznikiem klikniemy myszką. Stosowanie pozostałych rodzajów własności nie ma dla nas większego znaczenia.

Dla zainteresowanych - więcej szczegółów można znaleźć na przykład w podręczniku:

http://homeproject.pl/wp-content/uploads/2018/01/Podrecznik-HTML-CSS-2.pdf

Jednak powyższy zakres wiedzy naprawdę nam wystarczy aby zacząć przygodę z JavaScript.

Style

Atrybuty wyświetlania (na przykład kolor) można ustawiać za pomocą styli css. Na przykład `<p style="color:red;font-weight:bold">.`

Wróćmy do opisanych wcześniej ramek, aby pokazać jak za pomocą styli można określić ich atrybuty:

```
<div style="border: 1px solid blue;">
 ramka 1
</div>
<div style="border: 1px solid red;">
 ramka 2
</div>
<div style="border: 1px solid green;">
 ramka 3
</div>
```

W powyższym przykładzie opisano kolorowe obramowania. Podobnie możemy zmieniać teksty określone przez "span". Na przykład:

```
<span style="color:blue">
tekst niebieski
</span>
```

Aby ustawić je obok siebie - wykorzystuje się atrybuty szerokości i przepływu (float):

```
<div style="border: 1px solid;width:40%;float:left">
 ramka 1
</div>
<div style="border: 1px solid;width:40%;float:left">
 ramka 2
</div>
<div style="border: 1px solid;width:40%;float:left">
```

```
 ramka 3
</div>
<div style="clear:both;">
</div>
</div>
```

Aby odróżnić identyfikatory klas i znaczników poprzedza się je znakiem kropki (klasa) lub "płotka" (#). Dzięki temu możemy zapisać style w odrębnej sekcji html, opisując identyfikatorami to, co chcemy zmieniać:

```
<style>
 #ramka1 { background-color:"yellow"; }
 .kolorowe_teksty{font-weight:bold;}
</style>
```

Język opisu styli CSS jest bardzo bogaty. Nie ma sensu uczyć się wszystkich atrybutów na pamięć - gdyż istnieje wiele dostępnych on-line podręczników.

Podręczniki CSS:

https://pl.wikibooks.org/wiki/CSS

https://www.w3schools.com/css/default.asp(angielski, z dobrym tłumaczeniem Google)

JavaScript

Porównując bogactwo podręczników z powyższym krótkim opisem może to wydawać się dziwne, ale naprawdę powyższe informacje wystarczą do tworzenia stron internetowych - poza formularzami i elementami zmienianymi dynamicznie. Do tego służy właśnie JavaScript.

Mamy zatem trzy podstawowe elementy strony internetowej:

- treść: html
- wygląd (styl): css
- dynamikę: javascript

Wywołanie skryptu występuje wskutek wydarzenia - na przykład załadowania strony lub kliknięcie w jakiś jej element.

Kilka ważnych pojęć

Nawigacja

Zmiany w naszej przeglądarce najczęściej polegają na nawigacji – czyli przechodzeniu od strony do strony. Sterujemy tym używając przede wszystkim linków (url, znacznik a) oraz formularzy (znacznik form i input). Javascript umożliwia modyfikację strony bez jej przeładowywania.

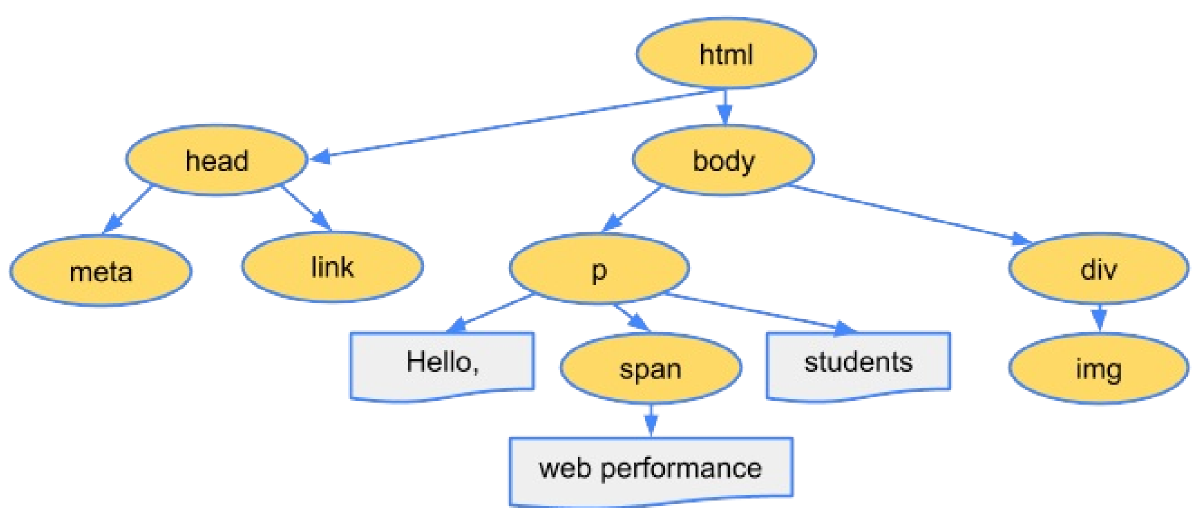
Renderowanie

Renderowanie – tworzenie wyglądu na podstawie opisu. Zazwyczaj pojęcie to odnosi się do tworzenia wyglądu przez przeglądarkę internetową.
Renderowanie w przeglądarce zaczyna się po pobraniu kodu HTML i CSS. W narzędziach dla webmasterów firmy Google (https://www.google.com/webmasters/tools/) można znaleźć „zobacz jako Google”, które pozwala sprawdzić, czy nasza strona jest tak samo widziana przez użytkownika jak i przez przeglądarkę. Jest tam opcja „renderuj”. Szczegółowy opis tego procesu (po angielsku): https://www.html5rocks.com/en/tutorials/internals/howbrowserswork/.

Drzewo DOM

Renderowanie w przeglądarce rozpoczyna się od zbudowania drzewa DOM na podstawie znaczników HTML. Drzewo – bo znaczniki są zagnieżdżane (tworzą strukturę drzewa).
Zobaczmy jak to wygląda dla prostej strony
(źródło):

```
  <html>
    <head>
      <meta charset="utf-8">
      <link href="style.css" rel="stylesheet">
      <title>Critical Path</title>
    </head>
    <body>
      <p>Hello <span>web performance</span>
  students!</p>
      <div><img src="awesome-photo.jpg"></div>
    </body>
  </html>
```



Obiekty

Każdy węzeł (node) tego drzewa (zaznaczone na żółto) jest obiektem. Czyli nie tylko jest to prosta wartość (jak liczba), ale zawiera własności (na przykład identyfikator). Na popielato zaznaczono jedną z własności - wyświetlaną treść.

Abstrakcyjne pojęcie obiektu jest obok zmiennej jednym z najważniejszych w informatyce. Zmienna to fragment pamięci komputera oznaczony nazwą. Obiektem jest struktura, która ma swoje własności (które także mogą być obiektami) i metody zmian tych własności (więcej:Programowanie : Uczymy_się myśleć abstrakcyjnie).

Ważne jest aby rozróżniać obiekt i klasę obiektów. Na przykład mamy obiekt Jana Kowalskiego klasy Student, która jest podklasą Ludzi. W przypadku drzewa DOM mamy klasy obiektów zdefiniowane przez znaczniki (np. div). Drzewo DOM jest obiektową strukturą strony internetowej (zob:https://pl.wikipedia.org/wiki/Obiektowy_model_dokumentu).

Uwaga terminologiczna.Nie należy kojarzyć klasy obiektów w drzewie DOM z własnością znaczników o nazwie**„class”.**Węzły drzewa**nie są**klasamidefiniowanymiprzez atrybut / własność znaczników**class**. Nazwa klasy znaczników (własność class) uważana jest na poziomie drzewa DOM za własność obiektów (węzłów drzewa). Przez analogię można to porównać do nazwisk ludzi. Nazwisko to własność każdego człowieka, ale pozwala też wyróżnić grupę ludzi o takim samym nazwisku. Gdyby w języku polskim zamiast słowo "nazwisko" funkcjonowało słowo "klasa", to mielibyśmy podobny kłopot z wyjaśnieniem pojęcia "klasy ludzi". Gdy mowa o DOM - można po prostu na chwilę zapomnieć o słowie "class" i traktować klasę jako rodzaj znacznika.

Program w JavaScript może odczytywać i zmieniać własności drzewa DOM.

Zdarzenia

Wiemy już do czego służy JavaScript. Jak jednak są uruchamiane programy napisane w tym języku? Są one wywoływane do obsługi zdarzeń. Na przykład najechania myszką na element strony lub kliknięcia w przycisk.

Zdarzenia obsługujemy skryptami napisanymi w JavaScript.

Funkcja

Absurdem byłoby, gdyby każde zdarzenie powodowało uruchamianie całego programu / skryptu zapisanego w Javascript. Dlatego naturalnym jest podział skryptu na drobne fragmenty służące do różnych celów. Fragmentom tym nadajemy najczęściej identyfikator, którym posługujemy się aby go wskazać. Taki fragment skryptu nazywamy funkcją. Ma on zapis następujący:

```
function identyfikator(parametry) {
  fragment skryptu
}
```

Typowy program Javascript może składać się z wielu funkcji. Funkcje nazywa się także czasem procedurami (w niektórych innych językach procedura to szczególny rodzaj funkcji – taka która nie zwraca żadnej wartości).

Identyfikatory

Aby móc odwoływać się do konkretnych obiektów, czy funkcji – musimy je jakoś nazywać. Do tego służą identyfikatory. Do ich tworzenia możemy użyć liter łacińskich (polskie znaki wykluczone), cyfr oraz znaków podkreślenia (_) I dolara ($). Pierwszym znakiem identyfikatora nie może być cyfra.

Przykłady identyfikatorów: `abc, dluga_nazwa, cyfra6, $pole.`

Nie są poprawnymi identyfikatorami napisy: `6a, dług, „pole”.`

Obiekty mogą mieć wewnętrzną strukturę składającą się z danych (zwanych własnościami) i funkcji (zwanych metodami). Odwołujemy się poprzez podanie ich identyfikatora po nazwie obiektu zakończonej kropką. Na przykład dla fikcyjnego obiektu reprezentującego człowieka możemy zdefiniować własność określającą jego wzrost: czlowiek.wysokosc, albo metodę wyświetlania informacji o nim: `czlowiek.informacje()` .

Obiekty i funkcje wbudowane

Jak już wiemy – skrypty działają w środowisku przeglądarki internetowej, a identyfikatory pozwalają wskazywać obiekty i funkcje. W jaki sposób jednak powiązać nasz opis (kod programu) z tym co dzieje się na stronie? Służą do tego elementy „wbudowane" - czyli takie, których nie musimy definiować (opisywać), bo one już istnieją w chwili uruchomienia programu. Mają one swoje identyfikatory – zdefiniowane przez twórców standardów (Javascript).

Tyle teorii nam wystarczy.

Więcej informacji:

- http://www.w3schools.com/html/html_intro.asp(można wybrać polskie tłumaczenie – jest dość porządne) .
- http://pdf.helion.pl/e14te3/e14te3.pdf(rozdział 3.8 i dalej).

Przejdźmy do pierwszego przykładu, który pokaże nam jak stosować funkcje do obsługi zdarzeń.

Pierwszy przykład

Istnieje wiele stron internetowych umożliwiających śledzenie (testowanie) działania skryptów w Javascript.Na przykład:http://codepen.io/pen/, https://jsfiddle.net/, http://jsbin.com/.Skorzystajmy z: http://jsbin.com/folowenasa/edit?html,css,js,console,output

Wpiszmy w treść strony jakiś tekst – na przykład „Kliknij tutaj".

Do znacznika `<body>` dodajmy własność „onclick" i nadajmy jej wartość „ `test()` ;". Będzie to znaczyć, że po kliknięciu na stronie wykona się procedura (funkcja) „test".

Przykład 1

```
<body onclick="test();">kliknij tutaj</body>
```

Procedura musi zostać napisana – żeby się wykonała:

```
function test() {
  alert('!');
}
```

Poniższe tabelki zawierają opis wszystkich symboli użytych w tym programie:

body	Część języka HTML w którym opisujemy strony internetowe. W tym wypadku znacznik używany do ograniczenia całej strony.	Oznacza wnętrze (treść) strony.
		Miejsce w tekście gdzie

< >	Dwa symbole określające początek i koniec znacznika.	zaczyna się i kończy znacznik.
/	Zmienia znaczenie znacznika (koniec oznaczanego obszaru) – o ile występuje natychmiast po znaku <	
onclick	Własność dodawana do znacznika. Po znaku = nadaje się tej własności wartość.	Co się stanie, gdy klikniemy na wskazywane przez znacznik miejsce.
function	Symbol używany do zdefiniowania fragmentu kodu (programu) nazywanego funkcją.	Definicję jakiejś funkcji.
test	Symbol zdefiniowany przez nas – konkretnej funkcji.	Definicję konkretnej funkcji test.
alert	Wyświetlenie komunikatu	Funkcję zdefiniowaną (wbudowaną) w przeglądarce.

Znaki używane dla interpretera (przeglądarki internetowej) – aby mógł jednoznacznie "zrozumieć" znaczenie tekstu:

Znak	**Znaczenie**
{	Początek bloku (funkcji)
}	Koniec bloku (funkcji)
;	Koniec instrukcji (z których budujemy funkcję).
"	Cudzysłów oznaczający początek i koniec napisu.
=	Znak równości (lub nadanie zmiennej określonej wartości)
(	Początek miejsca na parametry funkcji
)	Koniec miejsca na parametry funkcji

Jak widzimy - uruchomienie programu następuje wskutek jakiegoś zdarzenia. Tu wykorzystujemy zdarzenie „onclick”.

Znaczenie każdego symbolu zależy od kontekstu w którym go używamy. Słowo „alert” oznacza po angielsku alarm. W środowisku przeglądarki internetowej powoduje wyświetlenie komunikatu. Użycie symbolu w powyższym zdaniu jest jego przytoczeniem (cytatem). Po to używamy cudzysłowów. Jeśli w definicji funkcji test zmienimy alert(‘kliknąłem’) na alert(‘alert’) to nie wykona się dwa razy funkcja „alert”, tylko zostanie wyświetlona (przytoczona) jej nazwa.

Kontekst jest ważny, bo w różnych miejscach ten sam symbol może oznaczać coś innego. Nie należy tego nadużywać (lepiej stosować różne symbole).

Co się dzieje w przeglądarce internetowej

Najważniejszym zdarzeniem w przeglądarce jest załadowanie strony. Proces ten przebiega współbieżnie. Przeglądarka ściąga dane z serwera. Po ściągnięciu CSS i HTML rozpoczyna budowę drzewa DOM i renderowanie strony (dlatego mówi się, że CSS i HTML blokują renderowanie - proces rozpoczyna się po skompletowaniu CSS i HTML).
Po załadowaniu i zrenderowaniu strony jest ona wyświetlana – choć mogą być ściągane kolejne jej elementy (jak obrazki i skrypty JavaScript). Gdy przeglądarka uzna, że ma już komplet – wywołuje zdarzenie **onload**– na rzecz obiektu **document** (znacznik **body**). Zaleca się, aby skrypty które nie zawierają obsługi tego zdarzenia umieszczać nie w nagłówku, ale na końcu strony (wtedy **onload** zachodzi wcześniej - nawet gdy nie skończy się proces pobierania skryptów).

Poza onload najczęściej wykorzystuje się zdarzenie **onclick** wywoływane na rzecz obiektów z drzewa DOM (pełną listę darzeń umieszczono w dodatku A).

Przykład 2:

```
  <html>
    <head>
      <meta charset="utf-8">
  <script>
  function zdarzenie1() {
  alert("załadowałem");
  }

  function zdarzenie2() {
  alert("kliknąłem");
  }
  </script>
    </head>
   <body onload="zdarzenie1()">.
  <p>Kliknij
  <a href="#" onclick="zdarzenie2()">tutaj</a>
  </p>
    </body>
  </html>
```

Objaśnienie

Użyte w kodzie symbole:

Symbol	Znaczenie	oznaczanie
function	Symbol używany do zdefiniowania fragmentu kodu (programu) nazywanego funkcją (zob.	Definicję jakiejś funkcji.

	dalej).	
zdarzenie1 zdarzenie2	Symbole zdefiniowane przez nas – nazwy funkcji.	Definicje funkcji (zdarzenie1 i zdarzenie2).
alert	Wyświetlenie komunikatu	Funkcję zdefiniowaną (wbudowaną) w przeglądarce.

Przykład operacji na drzewie DOM

Aby nasz opis funkcjonowania skryptów na stronie internetowej był kompletny – musimy pokazać jak operować na drzewie DOM. Ten przykład może wydać się nieco trudniejszy. Tak jest w istocie – dlatego wprowadzono wiele ułatwień (jak tak zwaną bibliotekę jQuery o której będzie mowa dalej).

Obiekt dokumentu (document) posiada metody pozwalające odszukanie węzła drzewa. Na przyład na podstawie identyfikatora (własność id). Wykorzystamy to, by znaleźć znacznik o identyfikatorze „test" i odczytać jego kolor (część definicji stylu wyświetlania).

Przykład 3:

```
<html>
  <head>
  <script>
  function test() {
  alert(document.getElementById("test").style.color);
  }
  </script>
  </head>
  <body>
  <span style="color:blue"   id="test" onclick="test();"> kliknij tutaj
  </span>
  </body>
  </html>
```

W przykładzie tym kliknięcie w tekst powoduje wyświetlenie nazwy koloru. Możemy ten kolor też w JavaScript zmieniać:

```
document.getElementById("test").style.color = "red";
```

jQuery

Biblioteka (czyli dołączany z zewnątrz skrypt) jQuery nie wchodzi w skład definicji JavaScript, ale jest tak powszechnie używana, że można ją traktować jako standard. Dobre wprowadzenie do tej biblioteki można znaleźć na stronie: http://shebang.pl/kursy/podstawy-jquery/r3/

Do przećwiczenia przykładów możemy użyć serwisu jsbin.com.

Najważniejszą ideą jest całkowite oczyszczenie kodu HTML ze skryptów. Zamiast tego – po załadowaniu strony modyfikujemy drzewo DOM dodając w razie potrzeby obsługę zdarzeń.

Odwołanie do węzłów drzewa zwraca nam funkcja o nazwie $.

Zobaczmy to na prostym przykładzie .

Przykład 4a (bez jQuery):

```
<html>
  <head>
  <script>
    function zaladowalem() {
     document.getElementById("kolorowy1").style.color="green";
    }
  </script>
  </head>
  <body onload="zaladowalem()">.
    <p>Tekst <span id="kolorowy1" style="color:red">kolorowy</span></p>
 </body>
</html>
```

Serwis jsbin oozwala wyłączyć javascript – po prawej. Włączając lub wyłączając możemy zobaczyć tekst czerwony lub zielony.

Ten sam przykład z użycie jQuery (kod dołączenia z https://code.jquery.com):

Przykład 4b (z jQuery):

```
<html>
<head>
  <script src="https://code.jquery.com/jquery-3.2.1.min.js"></script>
  <script>
  function zaladowalem() {
    kolorowy1 = $("#kolorowy1");
    if (kolorowy1) {
    kolorowy1.css('color', 'green');
```

```
    }
  }
  $(document).ready(
     zaladowalem
  );
  </script>
</head>
<body>
  <p>Tekst <span id="kolorowy1" style="color:red">kolorowy</span></p>
</body>
</html>
```

Znacznik `<script>` z atrybutem src powala dołączać zewnętrzne pliki skryptów. Warto zapamiętać, że **nie jest dopuszczalny** skrócony zapis typu `<script src="..." / >` (bez znacznika kończącego).

Należy zwrócić uwagę na użycie # - tak jak w CSS. Tekst został specjalnie napisany tak, aby był łatwy do debugowania. Debugger (odpluskwiacz) pozwala na wykonywanie programu krok po kroku. Dla JavaScript dobrym debugger ma Firefox (narzędzia dla programistów –https://developer.mozilla.org/pl/docs/Narz%C4%99dzia/Debugger).

Powyższy przykład można uprościć używając funkcji anonimowych (tam gdzie parametrem może być funkcja, można także wpisać jej definicję – a nie nazwę jak w powyższym przykładzie):

```
<html>
  <head>
  <script
  src="https://code.jquery.com/jquery-3.2.1.min.js"></script>
  <script>
  $(document).ready(
    function() {
      $("#kolorowy1").css('color', 'green').
          css('font-weight', 'bold');
    }
  );
  </script>
  </head>
  <body>
  <p>Tekst
  <span id="kolorowy1" style="color:red">kolorowy</span>
  </p>
  </body>
  </html>
```

Objaśnienie:

1. Funkcja może nie mieć identyfikatora – jeśli wywołujemy ją tylko raz. Jest to tak zwana funkcja anonimowa.
2. Rezygnujemy ze sprawdzenia, czy obiekt kolorowy1 istnieje. Jeśli go nie ma – nie będzie błędu, tylko .css się nie wykona. To jedna z fajnych rzeczy w jQuery.
3. Wynikiem funkcji $ jest obiekt - ale nie obiekt drzewa DOM – dlatego nie odwołujemy się do własności style, tylko wywołujemy metodę css.
4. Tak samo obiektem jest wynik każdej metody wywoływanej na rzecz przetwarzanego obiektu. Dlatego możemy po kropce łączyć następne wywołania: `$("#kolorowy1").css('color', 'green').css('font-weight', 'bold');`

Dynamiczne budowanie strony w JavaScript

Podobnie jak metoda css zmienia styl, metoda html tworzy nowe fragmenty strony:

```
<html>
  <head>
  <script
  src="https://code.jquery.com/jquery-3.2.1.min.js"></script>
  <script>
  $(document).ready(function() {
  $("#kwadracik").html("<ul>dowolny fragment</u>
  strony").css('border', 'solid 1px green');
  });
  </script>
  </head>
  <body>.
  <div id="kwadracik"></div>
  </body>
  </html>
```

Jeśli parametrem funkcji $ nie jest selektor wyboru obiektów (np. identyfikator) ale HTML, to funkcja tworzy nowy obiekt. Można go później dołączyć w dowolne miejsce strony:

```
 <html>
  <head>
  <script
  src="https://code.jquery.com/jquery-3.2.1.min.js"></script>
  <script>
  var nowy = $('<div>drugi</div>');
  $(document).ready(function() {

  $("#kwadracik").html("<ul>dowolny fragment</u>  strony").css('border', 'solid 1px g
  nowy.appendTo('#kwadracik');
  });
```

```
  </script>
  </head>
  <body>
  <div id="kwadracik"></div>
  </body>
  </html>
```

Struktury, JSON

Jako parametr funkcji $().css() można podać kolekcję atrybutów do zmiany:

```
<html>
  <head>
  <script src="https://code.jquery.com/jquery-3.2.1.min.js"></script>
  <script>
  $(document).ready(function() {
  $("#kolorowy1").css({
  'color': 'green',
  'font-weight': 'bold'
  });
  });
  </script>
  </head>
  <body>
  <p>Tekst
  <span id="kolorowy1" style="color:red">kolorowy</span>
  </p>
  </body>
  </html>
```

Taka struktura ujęta w nawiasy klamrowe jest złożonym typem danych, który w JavaScript nazywa się obiektem. Definiujemy go podając rozdzielone przecinkiem pary: nazwa własnosci : wartość własności. Innym typem danych jest lista (tablica) – czyli zbiór wartości oddzielonych przecinkiem. Opisuje się go nawiasami kwadratowymi []:

```
var liczby = [1,2,3,4,5];
```

Więcej: http://shebang.pl/kursy/wszystko-jasne/r4-obiekty-tablice/

Opis struktur stosowany w JavaScript został przyjęty jako standard w komunikacji internetowej nazwany JSON. Należy jednak pamięać, że JSON to tekst, a nie obiekt. Napis '{"wlasnosc":"wartość"}' jest poprawnym tekstem JSON. Jeśli wczytamy go do zmiennej "zmienna", to nie uzyskamy możliwości dosępu do wartości poprzez zapis: zmienna.wlasnosc. Biblioteka jQuery zawiera funkcję parseJSON pozwalającą zapis (tekst) zamienić na strukturę:

```
<html>
  <head>
  <meta charset="utf-8">
  <script src="https://code.jquery.com/jquery-3.2.1.min.js"></script>
  <script>
    $(document).ready(function() {
    var zmienna = $.parseJSON( '{"wlasnosc":"wartość"}' );
    alert( zmienna.wlasnosc );
```

```
  });
  </script>
  </head>
  <body>.
  </body>
  </html>
```

Struktury

W komunikacji między aplikacjami można wyróżnić dwie sytuacje: klient jedynie umieszcza uzyskane dane na odpowiednim miejscu strony, lub przetwarza wcześniej ich zawartość.

W pierwszym przypadku dane mogą mieć strukturę strony (HTML). W drugim – muszą być w formie bardziej przyjaznej dla programów – na przykład JSON, XML lub SOAP.

Soap

Soap jest złożoną strukturą opracowaną przez firmę Microsoft. Opiera się ona na standardzie XML. Opis struktury może być wykonany w pliku **WSDL, który może służyć do automatycznej analizy komunikatów.**

JSON

Struktura SOAP jest zbyt złożona, aby ją analizować po stronie przeglądarki internetowej (w JavaScript). Dlatego opracowano standard Json, który dużo bardziej nadaje się do tego celu. Jest to struktura danych zamieniona na tekst. Na przykład lista (tablica) jest zamieniana na ciąg tekstów w nawiasach klamrowych: https://pl.wikipedia.org/wiki/JSON

Ajax

Terminem Ajax nazwano mechanizm przesyłania danych z serwera na stronę internetową, bez przeładowywania (wczytywania na nowo) całej tej strony.

Zobaczmy to na prosty przykładzie:

```
<html>
<head>
<meta charset="utf-8" />
<script  src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
<script>
$(document).ready(function () {
  $('#przycisk').click(function () {

   jQuery.ajax({
     url: 'example.txt',
     type: 'get',
     success: function(dane){
        $('#dane').append(dane);
     },
     error: function(xhr, status, error) {
       alert('Błąd');
     }
    });
  });    
});
</script>

</head>
<body>
<div id="dane"></div>
<button id="przycisk">kliknij</button>

</body>
</html>
```

Tak jak w poprzednich przykładach, mamy tu zdefiniowanie obsługi wydarzenia "onClick". W skrypcie wykorzystano funkcję ajax z biblioteki jQuery. Parametrem tej funkcji jest struktura (obiekt) o polach:

- **url** - adres pobierania (tu z pliku example.txt umieszczonym w tym samym miejscu na serwerze, co tworzona strona;
- **type** - sposób pobierania danych (get/post)
- **success** - finkcja wywoływana po pobraniu danych (parametrem są te dane)

- **error** - funkcja wywoływana w razie błedu

W funkcji "success" dodaje się wczytany tekst do sekcji div o identyfikatorze dane.

W tym przykładzie należy zwrócić uwagę na zrównoleglenie danych. Zapytanie jest wysyłane do serwera (funkcją ajax), natomiast funkcja success (;ub error) zostaje wywołana dopiero po zakończeniu transmisji. W międzyczasie strona / skrypt może wykonywać inne działania (równolegle do transmisji).

Funkcja takie jak error / success nazywamy funkcjami powrotu (**callback functions**).

Otrzymane dane mogą być w formacie Json - wtedy trzeba je przed użyciem przetworzyć. W jQuery istnieje funkcja specjalizowana do komunikacji Json: **$.getJSON** .

Przykład:

- Tworzymy plik Json z danymi:

```
{
  "items": [
    {
      "key": "Pierwszy",
      "value": 1
    },
    {
      "key": "Drugi",
      "value": 2
    }
  ],
  "title": "Dane testowe."
}
```

- Wczytujemy ten plik:

```
<html>
<head>
<meta charset="utf-8" />
<script  src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
<script>
$(document).ready(function () {

  $('#przycisk').click(function () {
    $.getJSON('example.json', function (data) {
      $('#tytul').append(data.title);

      var items = data.items.map(
        function (item) {
          return item.key + ': ' + item.value+'<br />';
        }
```

```
      );
      if (items.length) {
        $('#lista')
          .empty()
          .html('<li>' + items.join('</li><li>') + '</li>');
      }

    });
  });
});
</script>

</head>
<body>
<h1 id="tytul"></h1>
<ul id="lista"></ul>
<button id="przycisk">kliknij</button>

</body>
</html>
```

1) Zauważ, że funkcja powrotu (callback) jest drugim parametrem funkcja getJSON. Zwracane dane nie są tekstem, ale strukturą!!!

2) Listę do wyświetlenia przygotowuje metoda map (https://developer.mozilla.org/pl/docs/Web/JavaScript/Referencje/Obiekty/Array/map), która przekształca listę parametrów (z Json) na listę wartości wyliczonych na podstawie tych parametrów przez podaną funkcję. Tutaj funkcja tworzy napis `klucz: wartość` ;

3). Funkcja join (https://developer.mozilla.org/pl/docs/Web/JavaScript/Referencje/Obiekty/Array/join) łączy tablicę (listę) elementów w jeden napis (łańcuch znaków)

4). Zapis zaczynający się od `$('#lista')` króconą formą (często stosowaną) instrukcji:

```
if (items.length) {
  lista=$('#lista');
  lista.empty();
  lista.html('<li>' + items.join('</li><li>') + '</li>');
}
```

Taki skrót jest możliwy dzięki temu, że każda kolejna funkcja zwraca obiekt jQuery, który może być dalej przetwarzany.

API - wprowadzenie

API =Application Programming Interface

Jest to specyfikacja sposobu w jaki mogą być wymieniane dane między programami. Na przykład serwis internetowy może udostępniać API poprzez który inne serwisy mogą sięgać do danych, bez wyświetlania ich na stronie źródłowej.

Specyfikacja API wraz z przykładami i innymi materiałami przydatnymi programistom jest często udostępniana w postaci SDK:Software Development Kit.

Projekt API

Co wybrać? Gdy chcemy jedynie pobierać dane (na przykład poprzez Ajax). Wystarczy model pytanie – odpowiedź. Gdy użytkownik ma poprzez API działać na bazie danych – potrzebujemy REST lub usług sieciowych. Usługi sieciowe są niezbędne wówczas, gdykomunikacja ma bardziej złożony charakter. Stosowane są na przykład przy wysyłaniu deklaracji podatkowych.

Model pytanie – odpowiedź można traktować jako uproszczony REST, gdy chcemy jedynie pobierać dane. Stosując metodę GET podajemy parametry po prostu w adresie URL, łącząc je znakiem &. Na przykład:http://example.com/api?parametr1=wartosc1¶metr2=2

Model ten może także służyć do realizacji bardziej złożonych usług sieciowych, stanowiąc alternatywę dla bardziej złożonego SOPA. Przykładem może być usługa płatności PayU:http://developers.payu.com/pl/restapi.html

Zobacz też:

- http://www.moseleians.co.uk/wp-content/uploads/cmdm/9632/1422444257_api-restowe-whitepaper.pdf
- http://www.yiiframework.com/wiki/175/how-to-create-a-rest-api/

API - przykłady

Dla naszego przykładowego projektu (biblioteczki) można wykorzystać niektóre z dostępnych w sieci zasobów, dla których są dostępne odpowiednie API. Poniższy spis obrazuje też bogactwo zasobów, jakie można obecnie znaleźć w internecie:

Google Books:

http://www.programmableweb.com/news/53-books-apis-google-books-goodreads-and-sharedbook/2012/03/13

biblioteka:https://github.com/hubgit/libapi

Howto:

http://www.yiiframework.com/extension/ehttpclient/

http://www.yiiframework.com/wiki/697/make-an-ajax-request-to-another-server-using-jsonp/

OCLC:

https://pl.wikipedia.org/wiki/Online_Computer_Library_Center

https://www.oclc.org/developer/home.en.html

Wikipedia:

https://www.mediawiki.org/wiki/API:Main_page

https://github.com/thewulf7/Yii2-Wikipedia

LibraryThing

http://www.librarything.com/api

Opensearch: http://www.opensearch.org/Home

http://stackoverflow.com/questions/5216773/how-to-configure-yiis-urlmanager-for-opensearch-controller

Podstawy JavaScript

Pierwszy rozdział podręcznika to było prawdziwe “rzucenie na głęboką wodę”. Tak naprawdę nie poznawaliśmy w nim języka Javascript, ale sposób jego użycia. Jeśli ktoś chce jedynie w sposób świadomy korzystać z gotowych programów – może na tym wstępie poprzestać. Jednak naszym celem jest nauczenie się tworzenia własnych programów. Do tego potrzebujemy bardziej usystematyzowanej wiedzy. Ta część podręcznika jest inspirowana książką “Uczymy się programować w Pythonie”. Jednak zmiana języka ma szersze konsekwencje, niż tylko konieczność przepisania przykładów na nowo. Musimy zwracać niestety więcej uwagi na szczegóły techniczne – nie wiązane wprost z logiką programu. Wynika to wprost z przeznaczenia i strategii rozwoju języka. Javascript działa w środowisku przeglądarki internetowej, które ma swoje ograniczenia. Nie przewidziano na przykład prostych mechanizmów podawania danych z konsoli. Twórcy języka muszą także dbać o wsteczną zgodność, aby nie spowodować tego, że po wejściu w życie nowego standardu większość stron przestanie poprawnie się wyświetlać.

Konsola Javascript

Skrypty Javascript możemy wykonywać z konsoli przeglądarki. Na przykład w Google Chrome uruchamiamy ją klawiszem F12 (Windows / Linux) – zobacz: https://developers.google.com/web/tools/chrome-devtools/shortcuts.

Pomimo tego, że Javascript został stworzony z myślą o tworzeniu stron internetowych, możemy go używać poza środowiskiem przeglądarki internetowej. Na przykład wykorzystując oprogramowanie Nodejs. Projekt Nodejs zaowocował stworzeniem całego środowiska dla programistów Javascript. Warto go zainstalować na swoim komputerze – choć proste programy możemy testować na stronie:https://repl.it/ (po wybraniu środowiska nodejs). Przykłady z tej części podręcznika były testowane w konsoli **NodeJS**.

Po zainstalowaniu Nodejs w systemie Windows możemy uruchomić z menu „Node.js command prompt”. Uruchamia się konsola (wiersz poleceń Windows) z ustawieniami potrzebnymi do działania Nodejs. Następnym krokiem jest uruchomienie właściwej konsoli poleceniem node:

```
C:\Users\jurek>node
> .help
.break Sometimes you get stuck, this gets you out
.clear Alias for .break
.editor Enter editor mode
.exit Exit the repl
```

```
.help Print this help message
.load Load JS from a file into the REPL session
.save Save all evaluated commands in this REPL session to a file
> .exit
C:\Users\jurek>
```

Polecenie „.help” wyświetla dostępne komendy (poza instrukcjami Javascript). Wśród tych komend jest .exit – powodujący wyjście z konsoli.

W systemie Linux po zainstalowaniu nodejs można uruchamiać konsolę poleceniem node – bez konieczności robienia dodatkowych ustawień.

Konsola jest z punktu widzenia programu obiektem takim jak inne elementy drzewa DOM (https://www.w3schools.com/jsref/obj_console.asp). Najczęściej wykorzystuje się metodę **console.log()** - aby wyświetlić na konsoli jakąś informację.

Komentarze

Programy są wykonywane przez komputer, ale czytają je ludzie! Dlatego należy starać się pisać je w sposób przejrzysty, umieszczając komentarze w miejscach ewentualnych wątpliwości. Komentarze mogą mieć jedną z dwóch form:

1) Dwa ukośniki (//) oznaczają, że reszta wiersza nie zawiera instrukcji programu, ale komentarz.

2) Dłuższe komentarze zawiera się między znakami /* oraz */.

Przykład:

```
var pi=3.14; // przybliżona wartość liczby Pi


/* funkcja liczenia obwodu koła
 parametr: promień
 wynik: obwód */
function obwod( r ) {
 return 2*pi *r;
}
```

Dobra zasada: kod powinien sam się komentować. Zamiast pisać:

```
var kk = "red"; // kolor komunikatu
```

zapisz:

```
var kolorKomunikatu = "red";
```

Słowa kluczowe

Wybrane słowa są używane dla przekazania interpretatorowi Javascript

JavaScript jako kalkulator

Nowoczesne kalkulatory pozwalają na wpisanie ciągu znaków zawierających wyrażenie i wykonanie wszystkich działań za jednym razem. Tak samo może działać konsola Nodejs w której uruchamiamy polecenia JavaScriptu.

Większość instrukcji programu zawiera wyrażenia. Prostym przykładem wyrazem jest `2 + 3 .` Wyrażenie może być zbudowane z operatorów i argumentów.

Operatory to inaczej funkcje, które mogą być zapisane symbolami. Zamiast pisać `mnożenie(1,2)` zapisujemy `1*2` . Kolejność działań może być narzucona przez nawiasy. Na przykład: `(3+4)*3` . Dane na których są wykonywane działania (argumenty funkcji) nazywamy argumentami lub operandami (https://pl.wikipedia.org/wiki/Operand).

Operatory arytmetyczne

W JavaScript mamy dostępnych wiele różnych operatorów. Poniżej podano wykaz operatorów arytmetycznych:

Operator	Działanie	Przykład	Wynik
+	Dodawanie	3+2	5
-	Odejmowanie	3-2	1
*	Mnożenie	3*2	6
/	Dzielenie	3/2	1.5
%	Reszta z dzielenia	3 % 2	1

Uruchommy nodejs i spróbujmy:

```
$ node
> 3+2
5
> 3-2
1
> 3*2
6
> 3/2
1.5
> 3%2
1
```

Różne notacje liczb

W powyższym przykładzie występują liczby całkowite i ułamkowe - zwane w świecie informatyki zmiennoprzecinkowymi. Do zapisu takich lizb stosujemy kropkę dziesiętną (nie przecinek). Nie używa się znaków rozdzielających tysiące. Liczby te można zapisywać także w notacji wykładniczej. 1e4 oznacza 1*10 do potęgi 4:

```
> 1e4
10000
> 2e3
2000
> 123e3
123000
> 1e308
1e+308
> 1e308*10
Infinity
> 1/0
Infinity
```

Jak widać największa dopuszczalna jest liczba może mieć 309 cyfr. Próba jej zwiększenia prowadzi do błędu nieskończoności (Infinity). Tak samo jak próba dzielenia prez zero.

Zmienne

Podstawową różnicą między konsolą nodejs a kalkulatorem jest to, że w wyrażeniach JavaScriptu możemy używać zmiennych - czyli komórek pamięci w których przechowujemy wynik. Komórkom tym nadajemy nazwy - identyfikatory. W kalkulatorze też mamy namiastkę zmiennych: pamięć identyfikowaną literą M (od „memory”).

Zmienne w Javascript definiuje się słowem var (od variable), a od wersji ES6 także let i const (do tej kwestii jeszcze wrócimy).

Wstawienie wartości wyrażenia do zmiennej zapisuje się znakiem ‘=’.

Przykład:

```
> var x=10, y=12;
undefined
> x+y-1
21
> x = y % 5;
2
>
```

Konsola zakłada, że każda wykonana operacja zwraca jakąś wartość. Gdy nie jest to prawda – tak jak mamy to w pierwszym wierszu (jest tam tylko deklaracja zmiennych var….) - wyświetlana jest wartość „undefined”. W dalszej części podręcznika takie komunikaty będą pomijane.

Ten mechanizm wyświetlania danych nie należy do języka Javascript, ale jest sposobem działania konsoli. W samym języku jest oczywiście także polecenie wyświetlania dowolnych danych na konsoli:

```
> console.log(x);
2
```

W odniesieniu do zmiennych (pamiętających liczby) mamy dodatkowe dwie operacje arytmetyczne:

++	**Inkrement (zwiększenie o 1)**
--	Dekrement (zmniejszenie o 1)

Znaki `++` (`--`) ustawiamy przed lub po zmiennej, której zawartość chcemy zwiększyć (lub zmniejszyć). Gdy ustawimy je przed zmienną, operacja zwiększenia/zmniejszenia odbywa się przed użyciem w wyrażeniu.

Przykład:

```
> x=3
3
> y=x++
3
> x
4
> y=++x
5
> x
5
> x=x++
5
> x
5
> ++x
6
> y=(++x)*(++x)
56
>
```

Zauważ, że zapis `y = x++` jest równoważny z `++x` (zmienna y się nie zmienia). Co ciekawe - zapis `x=x++` nie powoduje żadnych zmian! To wcale nie jest oczywiste (co może być argumentem za tym, by takich konstrukcji nie (nad)używać). Celem takich zapisów jak w ostatnim wyrażeniu może być tylko utrudnienie zrosumienia ;-). Można traktować jako ciekawostkę.

Wyrażenia

Pokażemy użycie prostego wyrażenia do wyliczenia powierzchni prostokąta.

```
> length = 5; breadth = 2;
> area = length * breadth;
10
> console.log('Powierzchnia prostokąta = ', area);
Powierzchnia prostokąta = 10
```

Jak to działa

Długość i szerokość prostokąta są przechowywane w zmiennych odpowiednio length i breadth. Używamy ich do wyliczenia wyrażeń określających powierzchnię i obwód prostokąta. Wstawiamy do zmiennej area wynik wyrażenia length breadth, aby następnie wyświetlić go za pomocą funkcji console.log(druk). W tym przypadku obwodu możemy podać w funkcji drukowania bezpośrednio wyrażenie `2 (length + breadth)` – bez przechowywania wyniku w zmiennej.

Podstawowe operacje na napisach

Jak widzieliśmy w pierwszym rozdziale, w Javascript pozwala operować nie tylko na liczbach, ale także na napisach – zwanych ciągami znaków (ang. **string**). Ciąg znaków ujmuje się w cudzysłowu lub apostrofy. Podobnie jak liczby, ciągi znaków mogą być umieszczane w zmiennych i dodawane (łącone ze sobą).

Przykład:

```
var abc="Abecadło";
> var abc="ABC";
> console.log(abc+'22');
ABC22
```

Teoretycznie w obecnie używanych wersjach Javascript możemy zapisać także

```
> console.log(abc+22);
```

liczba zostaje przekształcona w napis i dołączona do napisu ze zmiennej abc.

Takie działanie nie jest jednak zalecane. Zamiast tego możemy jawnie przekształcić liczbę na napis, korzystając z tego że nawet proste dane (takie jak napisy i liczby) mogą być traktowane jako obiekt. Obiekt ten udostępnia między innymi funkcję (metodę) toString – zamieniającą wartość na łańcuch znaków (https://www.w3schools.com/js/js_number_methods.asp):

```
var abc="ABC";
console.log(abc+(22).toString());
ABC22
```

Ujęcie w nawiasy liczby jest konieczne, aby znaczenie kropki było jednoznaczne (22. jest poprawną liczbą).

Podobnie jako obiekt można traktować łańcuchy znaków. Dzięki temu uzyskujemy szereg metod manipulowania napisami: https://www.w3schools.com/js/js_string_methods.asp

Na przykład uzyskanie fragmentu łańcucha funkcją substring (https://www.w3schools.com/jsref/jsref_substring.asp):

```
> var abc="ABC";
> console.log(abc.substring(1,2));
```

```
B
```

Jak widać pierwszy parametr pokazuje początek wycinanego łańcucha (pozycja liczona od zera), a drugi (opcjonalny) – pozycję końca (a **nie długość** – co często ma miejsce w innych językach programowania).

Dygresja: nie ucz się wszystkiego na pamięć!

Powyższe proste przykłady wymagały pewnej wiedzy na temat struktury obiektów Number (liczba) i String (łańcuch znaków). Wiedza ta jest łatwo dostępna w internecie. Na przykład kompletne informacje o obiekcie liczby znajdziesz na stronach:

- https://www.w3schools.com/jsref/jsref_obj_number.asp
- https://www.w3schools.com/js/js_numbers.asp
- https://www.w3schools.com/js/js_number_methods.asp

W razie problemu – możesz szukać pomocy na stronach https://stackoverflow.com/.

Pełny opis Javascript zająłby tysiące stron – a na dodatek język ten dynamicznie się rozwija. Zmiany które dopiero są wprowadzane (wrócimy jeszcze do tego tematu) możesz znaleźć na stronie https://babeljs.io/learn-es2015/.

Celem nauki powinno być opanowanie języka i zasad pisania w nim programów. Po szczegółowe opisy dostępnych obiektów sięgamy wtedy, kiedy są potrzebne (taka metoda nauki nazywa się Just in time learning).

Instrukcje warunkowe i wartości logiczne

Kolejne instrukcje zapisane w języku JavaScript (jak i w każdym innym) zazwyczaj wykonują się w kolejności zapisu – sekwencyjnie: jedna za drugą. Jednak wiemy już, że działanie programu zależy od stanu obiektów. Nie zawsze ten stan wpływa jedynie na wynik wyrażenia. Na przykład chcąc uniknąć dzielenia przez zero – musimy sprawdzić wartość danych przed wykonaniem działania. Tego typu sprawdzenia wykonujemy przy użyciu instrukcji warunkowej, rozpoczynającej się od słowa **if**. Ma ona strukturę: if (warunek) blok_instrukcji. Blok instrukcji zaznaczamy nawiasami klamrowymi {}, a warunki nawiasami okrągłymi ().

Przykład:

```
> m=0;n=12;
> if (m == 0) {
console.log('Nie dziel przez zero');
} else {
wynik=n/m; console.log(wynik);
}
Nie dziel przez zero
```

Jeśli do zmiennej m wstawimy inną niż zero liczbę, uzyskamy wynik 12/m.

Instrukcja if służy do zaznaczenia bloku instrukcji, który się wykona wyłącznie wtedy, gdy prawdziwe będzie wyrażeni logiczne sprawdzane przed wejściem do tego bloku. Jeśli warunek jest spełniony, wykonywany jest blok instrukcji (możemy go określić jako "blok warunkowy"). w przeciwnym wypadku możemy wskazać inny blok instrukcji do wykonania. Służy do tego słowo (kluczowe) **else**. Użycie else jest opcjonalne.

Dwa znaki = (==) używamy w porównaniach dlatego, by nie mylić ich ze znakiem podstawienia (=). Choć to i tak jeden z najczęstszych błędów w programach.

Wartość logiczna

Wartość logiczna - jakiej używamy na przykład w instrukcji warunkowej - może zostać zapisana w zmiennej. Analogicznie jak w przypadku operacji arytmetycznych, język Javascript oferuje operatory do wyliczenia wartości logicznych:

- true – wartość logiczna prawda
- false – wartość logiczna fałsz

Wartość logiczną uzyskujemy przede wszystkim stosując operatory porównania:

- mniejszy <
- mniejszy lub równy <=
- większy >
- większy lub równy >=
- równy ==
- nierówny !=
- porównanie dokładne ===
- zaprzeczenie dokładnego porównania (różne) !==

Wyjaśnienia wymaga pojęcie dokładnego porównania. Widzieliśmy wcześniej, że Javascript potrafi zamienić liczbę na łańcuch znaków. Wartość prawdziwą zwraca więc wyrażenie:

`‘22’ == 22`

Porównanie dokładne zwraca prawdę jedynie wtedy, gdy wartości są równe bez dokonywania konwersji typów. Mamy zatem:

`‘22’ !== 22`

Javascript oferuje także operatory logiczne (algebry Boole’a):

- Zaprzeczenie (nie): !
- Alternatywa (lub): ||
- Koniunkcja (oraz): &&

To jednak nie wszystko. Aby uprościć sprawdzanie, czy jakiś obiekt lub własność istnieje i nie jest „pusta” - przyjęto, że następujące wartości są równoważne z false:

- `null –czyli wartość pusta`
- `undefined –własność nie zdefiniowana`
- `0 –liczba zero`
- `"" -pusty łańcuch znaków`
- `NaN –własność “to nie jest liczba`"

Mamy tu jak widać także wartości odnoszące się do obiektów i ich własności. Ich znaczenie stanie się jasne, gdy poznamy szczegóły programowania obiektowego.

A jakie wartości są traktowane jako prawda (true)? Tu obowiązuje prosta reguła: wszystkie, które nie są fałszem (false).

Przykład:

```
if ("") {
    console.log("wartość logiczna true");
} else {
    console.log("wartość logiczna false");
}
```

Zwraca: "wartość logiczna false"

Leniwa ewaluacja

Wyrażenia logiczne są wyliczane od lewej do prawej. Gdy tylko uda się ustalić wartość logiczną - dalsza ewaluacja (obliczanie wartości) wyrażenia nie jest kontynuowana.

Przyład:

```
> if (test()==0) console.log('!');
ReferenceError: test is not defined
> if ((1>0) || (test()==0)) console.log('!');
!
```

W pierwszej linii nastąpi błąd, gdyż funkcja test() nie istnieje. Nie można więc sprawdzić jej wyniku. Gdy przed tym sprawdzeniem pojawi się wyrażenie prawdziwe (1>0) połączone z resztą spójnikiem **lub** (||) - dalsa część wyrażenia nie jest sprawdzana.

Kolejność działań

W szkole nas uczono, że w wyrażeniach takich jak 2 + 3 * 4, najpierw robimy mnożenie. Mnożenie ma bowiem wyższy priorytet. W języku JavaScript również ustalono priorytety. Zaleca się jednak, aby nie polegać na tym, tylko stosować nawiasy (chodzi o jednoznaczność i czytelność zapisu).

W poniższej tabeli wymieniono poznane dotąd operatory w kolejności priorytetów - od największego do najmniejszego.

Operator	Opis
`++ -- - !`	Operatory jednoargumentowe, inkrementacja, dekrementacja, negacja, ...
`* / %`	Mnożenie, dzielenie, dzielenie modulo
`+ - +`	Dodawanie, odejmowanie, łączenie łańcuchów znaków
`< <= > >=`	Mniejsze niż, mniejsze lub równe, większe niż, większe niż lub równe
`&&`	AND logiczne

Najmniejszy priorytet ma `||` - OR logiczne (pominięte w tabeli ze względów technicznych).

Zmiana kolejności działań.

Aby uczynić wyrażenia bardziej czytelne, możemy użyć nawiasów. Na przykład, `2 + (3 * 4)` jest zdecydowanie łatwiejsze do zrozumienia niż `2+ 3 * 4` (ten zapiswymaga znajomości priorytetów operatorów. Jak ze wszystkim, nawiasy powinny być wykorzystywane racjonalnie – nie należy stosować ich w nadmiarze - jak w `(2 + (3+4))` .

Podstawową funkcją nawiasów jest jednak modyfikacja kolejności działań. Na przykład, jeśli chcesz wykonać dodawanie przed mnożeniem, to można napisać: `(2 + 3) * 4`.

Łączność

Operatory są zwykle stosowane z lewej do prawej. Oznacza to, że operatory o tym samym priorytecie są stosowane od lewej do prawej. Na przykład, `2 + 3 + 4` jest równoważny z `(2 + 3) + 4` .

Typy danych

Powyżej opisane zostały działania na wartościach liczbowych, znakowych i logicznych. Są to różne typy danych. Typ wartości jaką wstawimy do zmiennej decyduje o tym, jakiego typu jest to zmienna. Możemy to sprawdzić funkcją `typeof()` .

Przykład:

```
> var zmienna=1;
> typeof(zmienna);
'number'
> var zmienna='abc';
> typeof(zmienna);
'string'
> var zmienna=true;
> typeof(zmienna);
'boolean'
> var zmienna=undefined;
> typeof(zmienna);
'undefined'
> var zmienna=null;
> typeof(zmienna);
'object'
> var zmienna={};
> typeof(zmienna);
'object'
```

W przykładzie tym pokazano typy danych stosowane w Javascript. Widać z niego, że typ zmiennej może się zmieniać dynamicznie. Jest to tak wane **typowanie dynamiczne**.

W najnowszym standardzie Javascript (ES2015) wprowadzono możliwość typowania statycznego (zmienna nie może zmieniać typu). Dodano także jeden typ danych: Symbol. Na razie jednak zajmujemy się standardem obsługiwanym wprost przez większość przeglądarek….

Typy proste i referencje

Ostatnie dwa typy danych w poprzednim przykładzie dotyczą obiektów. Zapis zmienna={} powoduje utworzenie pustego obiektu. Ważne jest aby odróżnić pusty obiekt i wartość null (pustą).

Przykład:

```
> var pusty_obiekt={};
```

```
> var pusta_ref=null;
> pusty_obiekt.opis='dynamicznie stworzona własność';
'dynamicznie stworzona własność'
> pusta_ref.opis='??';
TypeError: Cannot set property 'opis' of null
>
```

W przykładzie tym widać, że do pustego obiektu możemy dynamicznie dodawać własności. Wiąże się z tym bardzo istotna kwestia: pusta wartość (null) należy do tak zwanych typów prostych. Do takich typów należą:

- boolean – wartość logiczna
- number - liczba
- string – łańcuch znaków
- undefined – wartość nie zdefiniowana
- null – wartość pusta
- symbol - symbol wprowadzony w standardzie ES2015

Przeciwieństwem do nich jest typ obiektowy: object. Fundamentalna różnica polega na tym, że zmienne typu object nie zapamiętują wartości (obiektów), tylko referencje (wskaźniki) do nich.

Różnicę wyjaśnia poniższy przykład:

```
> var obiekt1={};
> var obiekt2;
> obiekt2=obiekt1;
{}
> obiekt1.wlasnosc=12;
12
> obiekt2.wlasnosc;
12
> var liczba1=0;
> var liczba2;
> liczba2=liczba1;
0
> liczba1=12;
12
> liczba2;
0
>
```

Widać, że zmienne obiekt1 i obiekt2 wskazują na ten sam obiekt, gdy tymczasem liczba1 i liczba2 to zupełnie odrębne byty. Jeśli chcemy użyć zmiennych prostych jako obiektów, następuje ich konwersja do typu obiektowego, ale w zmiennej nadal pamiętana jest wprost wartość (a nie referencja).

Konwersja typów

Stałe i zmienne typów prostych są w użyciu zamieniane automatycznie na obiekty. Jeśli w wyrażeniach występują dane różnych typów, następuje ich konwersja - tak, by wyrażenie mogło być wykonane. Kryje się w tym pewne niebezpieczeństwo - ponieważ znak plus ('+') występuje zarówno w dodawaniu liczb jak i napisów - gdy nie ma jednoznaczności i Javascript przyjmuje, że chodzi o napisy!

```
> 2+2
4
> '2'+'2'
'22'
> 2+'2'
'22'
> 2+String('2')
'22'
> 2+Number('2')
4
> 2*'2'
4
> '2'/'2'
1
> true/String('2')
0.5
> (2).toString()
'2'
```

Ostatni przykład pokazuje, że rzeczywiście mamy do czynienia z obiektami i możemy wykorzystać ich własności. Opis tych własności możesz znaleźć na stronach:

- liczby https://developer.mozilla.org/pl/docs/Web/JavaScript/Referencje/Obiekty/Number
- łańcuchy znaków: https://developer.mozilla.org/pl/docs/Web/JavaScript/Referencje/Obiekty/String

Zamiast ujmować liczbę w nawiasy, można w miejsce kropki wpisać dwie kropki (chodzi o odróżnienie kropki dziesiętnej od kropki oznaczającej sięganie do środka obiektu:

```
> 2..toString()
'2'
```

Obiekty typu Number mają zdefiniowaną wartość NaN (Not a Number) - wskazującą na to, że to nie jest liczba

```
> 2 + undefined
NaN
```

```
>
```

Klasy i obiekty

W Javascript można zamiast o typach danych mówić o klasach tych danych - gdyż wszystkie zmienne i stałe są traktowane jak obiekty. Należy w tym miejscu zwrócić uwagę na różnicę między klasą a obiektem danej klasy. W tym drugim przypadku mówimy także o "instancjach" (wystąpieniach obiektu danej klasy). Niektóre metody operowania danymi są zdefiniowane dla klasy obiektów - i można ich użyć nawet gdy nie mamy żadnej instancji. Na przykład String.fromChaCode:

```
String.fromCharCode(65,66,67); // "ABC"
```

Ta funkcja zwraca łańcuch znaków na podstawie liczb oznaczających kody (Unicode - zgodne z ASCII) kolejnych liter.

Gdy tworzony jest obiekt danej klasy, wykorzystywany jest tak wany "prototyp". Obiekt otrzymuje wszystkie własności zdefiniowane w prototypie. Na przykład dla obiektów klasy string prototyp zawiera funkcję toUpperCase (zamiana na duże litery). Dla klasy String nie jest ona dostępna:

```
> ('błędy').toLocaleUpperCase();
'BŁĘDY'
> var msg='błędy'
> console.log(msg.toLocaleUpperCase());
BŁĘDY
> String.toLocaleUpperCase();
TypeError: String.toLocaleUpperCase is not a function
```

Wszystkie metody prototypu łańcuchów znaków znajdziesz na stronie:
https://developer.mozilla.org/pl/docs/Web/JavaScript/Referencje/Obiekty/String/prototype

Jedną z najciekawszych cech Javascript jest możiwość dodawania do prototypu własnych własności i metod. Nowo powstałe obiekty (instancje) danej klasy otrzymują tak dodane cechy:

```
> String.prototype.hash='#';
'#'
> var s1=String('abc')
> console.log(s1.hash);
#
```

Funkcje i parametry

Znamy ze szkoły funkcje takie jak prosta parabola: f(x)=x*x. W językach programowania rozróżnia się definicję i stosowanie funkcji. Jak już pokazano we wcześniejszych przykładach, definicja funkcji rozpoczyna się od słowa function, po którym może wystąpić nazwa, parametry i blok kodu.

Użycie (wywołanie) funkcji polega na podaniu jej nazwy i parametrów. W tym przypadku:

```
> function f(x) { return x*x; }
> f(12);
144
> a=11;
11
> f(a)
121
> b=f(2);
4
> f(b);
16
```

Słowo return oznacza wynik jaki funkcja zwraca. Wynik ten może być użyty w dalszych działaniach. Na przykład wstawiony do zmiennej – jak w wierszu `b=f(2)`.

Warto też zwrócić uwagę na to, że wartości zmiennych (a,b) przekazane do funkcji są dostępne wewnątrz funkcji pod nazwą x (czyli nazwą parametru).

Parametrów może być więcej niż jeden. Rozdzielamy je wówczas przecinkami.

Funkcje w językach programowania tylko w pewnym stopniu działają tak, jak w matematyce. Najistotniejszą różnicą jest to, że we wnętrzu funkcji możemy sięgać do wartości z otoczenia w którym funkcja została umieszczona (nie zawsze więc dla tych samych wartości parametrów uzyskamy taki sam wynik).

Przykład:

```
> var przesuniecie = 0;
> function f(x) { return x*x+przesuniecie; }
> f(3);
9
> przesuniecie=10;
> f(3);
19
```

Taki sposób programowania nie jest zalecany, gdyż prowadzi do pomyłek. Lepiej przekazać wszystkie używane wartości w postaci parametrów:

```
> function f(x,przesuniecie) { return x*x+przesuniecie; }
> f(3,10);
19
> f(3,5);
14
```

Zakres widoczności identyfikatorów

Jednym z problemów jakie są związane z dostępem do środowiska poza funkcją są tak zwane **domknięcia** (closures). Zasadniczo chodzi o to, że zmienna jest widoczna w bloku funkcji w której ją zdefiniowano. Definiując nową zmienną o takiej samej nazwie – przykrywamy poprzednią, ale tylko w obrębie funkcji w której dokonano "przykrycia".

Przykład:

```
> var przesuniecie=1;
function f(x) { var przesuniecie=11; return x*x+przesuniecie; }
> f(3)
20
> przesuniecie=10;
10
> f(3)
20
> przesuniecie;
10
>
```

UWAGA! Zadeklarowana słowem **var** zmienna jest widoczna w całej funkcji - także w instrukcjach poprzedzających wspomianą deklarację.

W najnowszej wersji standardu Javascript (ES6) wproadzono alternatywne mechanizmy deklarowania zmiennych. Zamiast słowa var można użyć `let` (lokalna definicja zmiennej) lub `const` (lokalna definicja zmiennej, która nie zmienia swej wartości). Lokalna – czyli taka, która jest widoczna wyłącznie w bloku w jakim ją definiowano (nie musi to być blok funkcji).

Przykład:

```
> if (true) { var x=1; }
> x
1
> if (true) { let y=1; }
> y
```

```
ReferenceError: y is not defined
> if (true) { const z=1; }
> z
ReferenceError: z is not defined
```

Skutki uboczne

Wiemy już, że zmienne przechowują referencje do obiektów, a nie tylko obiekty. Dotyczy to także parametrów. Dzięki temu skutki działania funkcji mogą wykraczać poza zwracanie wyniku poprzez return.

Przykład:

```
var obiekt2={};
var liczba2=1;
> function test_ref(o,l) {
 o.wlasnosc='zmiana';
 l=999;
}
> test_ref(obiekt2,liczba2);
> obiekt2;
{ wlasnosc: 'zmiana' }
> liczba2;
1
```

UWAGA! Jeśli zadeklarujemy obiekt2 jako `cons` (zamiast `var`) – efekt będzie taki sam. Użycie const mówi o tym, że niezmienna jest referencja do obiektu. Jego wnętrze może ulegać zmianie.

Instrukcje i algorytmy

Komputer działa w sposób deterministyczny – to znaczy, że zawsze w danym stanie (zawartość pamięci i urządzeń wejściowych) zachowa się tak samo. Jednak wspomniany stan może opisywać dane wejściowe, w zależności od których program musi zrealizować różne warianty działania. Służą do tego wspomniane wcześniej instrukcje warunkowe. Na przykład:

```
  jeśli a>b to napisz(‘a>b’);
  jeśli a<b to napisz(‘a<b’);
```

W językach programowania w miejsce „jeśli” pojawia się angielskie „if”.

Przykład:

```
 <form>
  rok= <input id="rok" type=text value="2016"><br />
  miesiąc= <input id="mies" type=text value="01"><br />
  <input type="button" onclick="javascript:licz()"  value="licz"/><br >
  dni= <input id='dni' type=text value="00" readonly/><br />
  </form>
```

Javascript:

```
function licz(){
     rok=document.getElementById('rok').value;
     mies=document.getElementById('mies').value;
     var dni='31';
     if (mies=='02') {
      dni='28';
      if (rok==2016) { dni='29'; }
    }
    if (mies=='04') { dni='30'; }
    if (mies=='06') { dni='30'; }
    if (mies=='09') { dni='30'; }
    if (mies=='01') { dni='30'; }
    document.getElementById('dni').value=dni;
  }
```

Poza instrukcjami warunkowymi, do zapisu algorytmów potrzebujemy pętli - czyli powtarzania bloku instrukcji. Javascript oferuje kilka rodzajów pętli.

1) Chyba najczęściej spotykaną pętlą jest instrukcja for/in. Służy ona do "przeglądania" wszystkih elementów struktury lub listy.

Przykład:

```
<button onclick="petla_for_in()">Kliknij</button>

<p id="wynik"></p>

<script>
function petla_for_in() {
    var struktura = {name:"Jan", age:20};
    var tablica = ['a','b','c'];
    var text = "";
    for (x in tablica) {
        text += x + ': '+tablica[x] + "  ";
    }
    var x;
    for (x in struktura) {
        text += '<br />' + x + ': '+struktura[x];
    }
    document.getElementById("wynik").innerHTML = text;
}
</script>
```

Po kliknięciu uzyskujemy wynik:

```
0: a 1: b 2: c
name: Jan
age: 20
```

Dla osób znających inne języki w któych pętla for/in występuje może to być pewna niespodzianka (w najnowszej wersji standardu wprowadzono for/of działającą w taki jak spodziewany sposób). Używa w tym przykładzie zmienna x przyjmuje w pętli wartość "indeksu" - czyli elementu pozwalajacego wybrać własność (indeks tablicy asocjacyjnej). Przy okazji pokazano, że w Javascript zadeklarowana przy pomocy **var** zmiennajest widoczna w całej funkcji (nawet przed deklaracją).

2) For w stylu C - czyli jedna z najstarszych instrukcji w językach programowania wygląda następująco: for (inicjowanie indeksu; warunek zakończenia; zwiększanie indeksu). Przykład:

```
<button onclick="sumuj()">kliknij</button>

<p id="wynik"></p>

<script>
```

```
function sumuj() {
    var suma = 0;
    var i;
    for (i = 0; i < 8; i++) {
        suma += (i+1)*2;
    }
    document.getElementById("wynik").innerHTML = suma;
}
</script>
```

W tej pętli liczona jest suma pierwszych 8 liczb parzystych. Indeksem zwyczajowo jest i (co oczywiscie nie jest konieczne).

3) Powyższą pętlę można zapisać prościej przy pomocy instrukcji **while**. Ma ona strukturę: while (warunek powtarzania):

```
<button onclick="sumuj()">kliknij</button>

<p id="wynik"></p>

<script>
function sumuj() {
var suma = 0;
var i=0;
while (i < 8) {
  suma += (i+1)*2;
  i++;
}
document.getElementById("wynik").innerHTML = suma;
}
</script>
```

Jak widać - przy użyciu tej pętli musimy zadbać o to, aby warunek powtarzania był ustawiony na true i osiągnął wartość false - inaczej pętla się nie wykona lub będzie trwała w nieskończoność. Dlatego *w tym przykładzie) ustawiamy zmienną i przed pętlą oraz zmieniamy na końcu.

Z tego powodu (ryzyka błędów) ta pętla używana jest dużo rzadziej - tylko wtedy, gdy na początu nie możemy prosto ustalić ilości powtórzeń.

5) Jeszcze radziej spotyka się pętlę do/while - z warunkiem sprawdzanym na końcu (https://www.w3schools.com/jsref/jsref_dowhile.asp)

Callback

Funkcje powrotu (callback) służą zrównolegleniu operacji. Weźmy prosty przykład odczytywania danych z pliku:

```
var fs = require("fs");
var data = fs.readFileSync('data.txt');
console.log(data.toString());
```

Uzyto w nim funkcji readFileSync, która zwraca przczytane dane. W czasie odczytu nic innego nie moze być przez program realizowane.

```
var fs = require("fs");

fs.readFile('data.txt', function (err, data) {
   if (err) return console.error(err);
   console.log(data.toString());
});console.log("Dalsze działania....");
```

Funkcja readFile umożliwia podanie w drugim parametrze funkcji powrotu (callback), ktora zostanie wywołana po zakonczeniu odczytu. Program nie czeka na zakończenie czytania, ale realizuje kolejne operacje. Jeśli plik data.txt zawiera jeden wiersz: 'testowe dane', wykonanie drugiego skryptu spowoduje wyjście na konsoli:

```
Dalsze działania....
testowe dane
```

Rozwój języka

- Funkcje, klasy i obiekty
- Dialekty
- ES2015
- TypeScript

Funkcje, klasy i obiekty

Klasyczny Javascript nie jest językiem w pełni obiektowym, choć zmienne a nawet stałe (napisy i liczby) i funkcje zachowują się jak obiekty. Jednak nie można w prosty sposób tworzyć obiektów pochodnych (dziedziczenie) ani definiować klas obiektów. Wykorzystujemy do tego funkcje, które mogą udawać klasy (obiekt wzorcowy) lub obiekty. Możemy utworzyć egzemplarz takiej klasy (funkcji wzorcowej), który jest obiektem. Wykorzystywany jest do tego tak zwany prototyp.

Klasa w JavaScript wygląda mniej więcej tak:

```
var NazwaKlasy = (function () {
 function NazwaKlasy() {
 }
 NazwaKlasy.prototype.jakasWlasnosc = function () {
 };
 return NazwaKlasy;
})();
```

Sprawdźmy:

```
> var nazwaKlasy = (function () {
   function Konstuktor() { }
   return Konstuktor;
 })();
> var obiekt= new nazwaKlasy();
> typeof(nazwaKlasy);
'function'
> typeof(obiekt);
'object'
```

Polecenie: `new Klasa()` tworzy nowy obiekt danej klasy.

Przykład:

```
> var Zwierze = (function () {
 function Zwierze() {
 }
 Zwierze.prototype.rodzaj = function () {
 return 'zwierze';
 };
 return Zwierze;
 })();
```

```
> var obiekt = new Zwierze();
> console.log(obiekt.rodzaj());
zwierze
```

Na szczęście takie klasy są tworzone głównie przez twórców bibliotek. W codziennym programowaniu częściej mamy do czynienia z konkretnymi obiektami, których użycie jest łatwe.

Słowo this

Słowo kluczowe **this** jest używane do wskazania na obiekt właściciela funkcji w której to słowo stosujemy. W języku JavaScript działa ono inaczej, niż mogą tego oczekiwać osoby znające inne – obiektowe języki programowania. Wynika to z faktu, że w Javascript to funkcje „udają” obiekty.

Zdefiniujmy klasę z własnością "komunikat" i metodę o nazwie "metoda":

```
var NazwaKlasy = (function () {

  function NazwaKlasy(komunikat) {
    this.komunikat = komunikat;
  }

  NazwaKlasy.prototype.metoda = function () {
    return "Moja własność: " + this.komunikat;
  };

 return NazwaKlasy;

})();


var instancja = new NazwaKlasy("witaj");

console.log(instancja.komunikat);
console.log(instancja.metoda());
instancja.komunikat='zmiana';
console.log(instancja.metoda());
```

Na wynik otrzymamy:

```
witaj
Moja własność: witaj
Moja własność: zmiana
```

Wszystko w miarę jasne. Co jednak się stanie, gdy definiowane metody będą bardziej złożone?

```
var NazwaKlasy = (function () {

  function NazwaKlasy(komunikat) {

    this.komunikat = komunikat;

  }

  NazwaKlasy.prototype.metoda1 = function () {
    return "Moja własność: " + this.komunikat;
  };


  NazwaKlasy.prototype.metoda2 = function () {
    function wyswietl() {
      return "Moja własność: " + this.komunikat;
    };
    return wyswietl();
  };

  return NazwaKlasy;

})();


let instancja = new NazwaKlasy("witaj");..

console.log(instancja.metoda1());
console.log(instancja.metoda2());
```

Na wynik dostaniemy:

```
Moja własność: witaj
Moja własność: undefined
```

Dlaczego?

Bo słowo this odnosi się do "właściciela" - a dla funkcji "wyswietl" właścicielem jest "metoda2"!

Aby uniknąć takich problemów wprowadza się dodatkową zmienną (własność) zwyczajowo nazywaną "self":

```
var NazwaKlasy = (function () {

  function NazwaKlasy(komunikat) {

    this.komunikat = komunikat;

  }
```

```
  NazwaKlasy.prototype.metoda1 = function () {
    return "Moja własność: " + this.komunikat;
  };


  NazwaKlasy.prototype.metoda2 = function () {
    var self = this;

    function wyswietl() {
      return "Moja własność: " + self.komunikat;
    };
    return wyswietl();
  };

  return NazwaKlasy;

})();


let instancja = new NazwaKlasy("witaj");..

console.log(instancja.metoda1());
console.log(instancja.metoda2());
```

Teraz otrzymujemy wynik zgodny z oczekiwaniem.

Bind

Jedną z konsekwencji używania w funkcjach `this` jest to, że metody obiektu nie można bezproblemowo oderwać od obiektu.Dotyczy to na przykład przykład obiektu console. **Poniższy kod jest błędny!!!**:

```
  var log = console.log;
  log('ABC');
```

Funkcja `log` zakłada, że `this` odnosi się do `console` , ale to jest zachowane wyłącznie wtedy, gdy stosujemy notację kropkową - sięgając wprost do wnętrza obiektu (`console.log`). Można ten problem rozwiązać, stosując metodę bind - wiążącą funkcję z właściwym obiektem (funkcja this będzie wskazywać poprawny obiekt):

```
  var log = console.log.bind(console);
  log('ABC');
```

Struktura a funkcja

Określenie "obiekt" używane jest w odniesieniu do struktury ({}) jak i pamiętanej w zmiennej funkcji. Czy to nie prowadzi do pomyłek? Nie - ponieważ na najbardziej podstawowym poziomie obiekt w języku JavaScript ma zawsze postać "tablicy asocjacyjnej" złożonej z kluczy i wartości. Możesz nawet sam się o tym przekonać, uzyskując dostęp do własności obiektu za pomocą składni tablicy. Oto przykład:

```
let obiekt1 = { nazwa : 'Obiekt 1'};
let obiekt2 =  function () { }
obiekt2.nazwa ='Obiekt 2';
console.log(obiekt1["nazwa"]);
console.log(obiekt2["nazwa"]);
```

W obu przypadkach dostaniemy na wynik właściwą nazwę obiektu. Z uwagi na prostszą składnię wygodniej jest oczywiście używać struktur.

Nowy standard

Jak widać użycie obiektów jest proste. Jednak stosowanie klas już jest mocno pogmatwane….

Nic więc dziwnego, że nowsze dialekty Javascript wprowadzają zmiany, dzięki którym klasy i obiekty przypominają to, co znamy z innych języków programowania.

Standard ECMAScript 2015 (ES2015) oferuje tak zwany „lukier składniowy”, czyli proste zapisy kryjące nieco bardziej złożone mechanizmy. Prosty przykład obiektów w ES2105:

```
class Empty {
}
class HelloWorld extends Empty {
  render() {
  return"Hello World";
}
}
var instance1 = new HelloWorld();
console.log(instance1.render());
```

Po przetworzeniu translatorem Babel uzyskujemy czysty Javascript.

Fragmenty wyniku tej translacji widzimy poniżej:

```
"use strict";
```

```
var _createClass = function() {
function defineProperties(target, props) {
for (var i = 0; i < props.length; i++) { var descriptor = props[i];
descriptor.enumerable = descriptor.enumerable || false;
descriptor.configurable = true; if ("value"in descriptor) descriptor.writable = true;
Object.defineProperty(target, descriptor.key, descriptor);
}
}
returnfunction(Constructor, protoProps, staticProps) {
if (protoProps) defineProperties(Constructor.prototype, protoProps);
if (staticProps) defineProperties(Constructor, staticProps);
return Constructor;
};
}();


function _possibleConstructorReturn(self, call) {
if (!self) {
thrownew ReferenceError("this hasn't been initialised - super() hasn't been called");
}
return call && (typeof call === "object" || typeof call === "function") ? call : self
}


function _inherits(subClass, superClass) {
if (typeof superClass !== "function"&& superClass !== null) {
thrownew TypeError("Super expression must either be null or a function, not " +
          typeof superClass);
}
subClass.prototype = Object.create(superClass && superClass.prototype, {
constructor: {
value: subClass,
enumerable: false,
writable: true,
configurable: true
}
});
if (superClass) Object.setPrototypeOf ?
Object.setPrototypeOf(subClass, superClass) : subClass.__proto__ = superClass;
}


function _classCallCheck(instance, Constructor) {
if (!(instance instanceof Constructor)) { thrownew TypeError(
   "Cannot call a class as a function"); }
}


var Empty = function Empty() {
_classCallCheck(this, Empty);
};
```

```
var HelloWorld = function(_Empty) {
_inherits(HelloWorld, _Empty);


function HelloWorld() {
_classCallCheck(this, HelloWorld);


return _possibleConstructorReturn(this,
(HelloWorld.__proto__ || Object.getPrototypeOf(HelloWorld)).apply(this, arguments));
}


_createClass(HelloWorld, [{
key: "render",
value: function render() {
return"Hello World";
}
}]);


return HelloWorld;
}(Empty);


var instance1 = new HelloWorld();


console.log(instance1.render());
```

Uff - łatwo nie jest ;-). Na szczęście nie musimy wnikać w te szczegóły implementacyjne.

Nieco prościej robi to TypeScript (https://www.typescriptlang.org/play/index.html):

```
var __extends = (this&&this.__extends) || (function () {
// przepisanie własności przodka
returnfunction (d, b) {
extendStatics(d, b);
function __() { this.constructor = d; }
d.prototype = b === null ? Object.create(b) : (__.prototype = b.prototype, new __());
};
})();


var Empty = /**@class*/ (function () {
function Empty() {
}
return Empty;
}());
```

```
var HelloWorld = /**@class*/ (function (_super) {
__extends(HelloWorld, _super);
function HelloWorld() {
return _super !== null&& _super.apply(this, arguments) || this;
}
HelloWorld.prototype.render = function () {
return"Hello World";
};
return HelloWorld;
}(Empty));
```

__extends Odpowiada ona za kopiowanie funkcji z prototypu podstawowego do klasy potomnej

Dialekty

Niedogodności Javascript poprawiają jego dialekty, zwane czasem „nadzbiorami” takie jak ES2015, React, czy TypeScript. Wszystkie one konwertują się do standardowego Javascript.

Typescript został opracowany przez Microsoft, React przez Facebooka, ES2015 to nowy standard języka. Wszystkie te trzy "dialekty" mają ze sobą wiele wspólnego. React to rozszerzenie ES2015 o JSX. Natomiast Typescript wprowadza możliwość definiowania własnych typów danych.

Nowe wersje wspomnianego wcześniej Angulara (stworzone przez Google framework przeznaczony jest do tworzenia interaktywnych stron Single-Page-Application, SPA) zostały zaimplementowane z użyciem Typescript..

ES2015

Poniższy opis standardu ES6 (ES2015) jest jedynie krótkim przeglądem (inspirowanym przez https://babeljs.io/learn-es2015/). Zainteresowanych bardziej szczegółowym opisem odsyłamy do podręcznika Nicholasa C. Zakasa "Ecmascript 6. Przewodnik po nowym standardzie języka Javascript".

let i const

Jak już wspomniano, zmienne zadeklarowane słowem **var** - są widoczne w całym bloku funkcji niezależnie od miejsca w którym zadeklarowano (hoisting)!!!

Jako alternatywę wprowadzono dwa słowa kluczowe: let i const.

Zmiennych zadeklarowanych z ich pomocą nie można deklarować ponownie (deklarować zmiennej o tej samej nazwie). Czyli ogranicza to dynamiczne typowania.

Zmienna **const** musibyć dodatkowo zainicjalizowana - i to tylko raz (nie zmienia wartości).

Najważniejsze jest to, że tak zadeklarowane zmienne są widoczne tylko w bloku w którym je zadeklarowano (nie tylko w bloku funkcji - ale na przykład w pętli). Błąd nie zostanie więc zgłoszony w sytuacji, gdy deklaracja let utworzy nową zmienną o takiej samej w blokach nadrzednym i podrzednym (np. w funkcji i pętli wewnątrz tej funkcji). To będą dwie różne zmienne o tej samej nazwie.

Użycie deklaracji const uniemożliwia modyfikację wiązania (wartości zmiennej), ale nie wartości obiektu na jaki wskazuje.

Dostęp do zmiennej zadeklarowanej za pomocą słów kluczowych let lub const nie jest (w przeciwiństwie do var) możliwy przed pojawieniem się deklaracji:

```
> if (typeof(v) != 'undefined') {console.log('zmienna widoczna');} var v = "wartosc";
zmienna widoczna
> if (typeof(l) != 'undefined') {console.log('zmienna widoczna');} let l = "wartosc";
ReferenceError: lzmienna is not defined
>
```

Przykład:

```
> if (true) {
  let zmienna = "wartosc";
  const stala={v: 1};
  stala.v=2;
  console.log(stala.v);
  console.log(zmienna);
 }
2
wartosc

> console.log(stala.v);
ReferenceError: stala is not defined
> console.log(zmienna);
ReferenceError: zmienna is not defined
>
```

Zaleca się stosowanie wszędzie, gdzie to tylko możliwe - deklaracji **const**.

Pętla for/of

W standardzie ES6 wprowadzono pętlę for/of jako alteratywę do for/in:

```
<button onclick="petla_for_of()">Kliknij</button>

<p id="wynik"></p>

<script>
function petla_for_of() {
    var lista = ['a','b','c'];
    var text = "";
    for (let x of lista) {
        text += x + "  ";
    }
    document.getElementById("wynik").innerHTML = text;
}
</script>
```

Wynik:

```
a b c
```

W tej pętli indeks (x) przyjmuje wartości z listy (a nie klucze jak w pętli for/in). Przy okazji pokazano jak w typowy sposób deklarujemy w pętli zmienne przy pomocy "let" .

Funkcje strzałkowe

Zamiast function(parametry){instrukcje} możemy zapisać (parametry)=>{instrukcje}. Przyczym nawiasy są obowiązkowe tyko wtedy, gdy ich brak prowadzi do niejednoznaczności.

Przykład:

```
let lista=[1,2,3,5];
let suma=0;
for (n of lista.map( (x)=> {return x*x*x;} ) ) suma+=n;
console.log(suma);
```

```
// wersja uproszczona:
let lista=[1,2,3,5];
let suma=0;
for (n of lista.map( x =>  x*x*x ) ) suma+=n;
console.log(suma);
```

Wynik: 161

Funkcja użyta w jako parametr dla map liczy sześcian liczby. Mamy więc tutaj sumę szcześcianów.

Poza uproszczeniem zapisu, dodatkową (a może główną) zaletą funkcji strzałkowych jest to, że słowo **this** zachowuje się w nich w sposób bardziej intuicyjny - funkcje strzałkowe nie zmienają jego znaczenia. A więc użyte wewnątrz klasy/obiektu zagnieżdżone funkcje stale odnoszą się do tego smego this.

Przykład:

```
let lista=[1,2,3,5];
let suma=0;
const licz = () => {
  for (n of lista.map( x =>  {this.suma+=x*x*x} ) );
  console.log(suma);
};
licz()
console.log(suma);
```

Na wynik dostaniem dwukrotnie 161.

Wartość domyślna parametrów

W ES6 wprowadzono wartości domyślne parametrów funkcji. Na przykład:

```
function kwota(ile, waluta='PLN', przed=false) {
  return przed ? waluta+ile.toString() : ile.toString()+waluta;
}
console.log(kwota(12));
console.log(kwota(10,'$',true));
```

Możemy przy wywołaniu funkcji podać tylko część parametrów.

Uwaga: opuszczenie części parametrów, a podanie następnych może prowadzić do błędów:

```
> console.log(kwota(10,przed=true));
10true
```

Klasy i obiekty

Przykład:

```
class Osoba {
    constructor(imie,nazwisko){
        this.imie=imie;
        this.nazwisko=nazwisko;
    }
    przedstawSie(){
        console.log(this.imie+' '+this.nazwisko);
    }
}

class Pracownik  extends Osoba{
    constructor(nr,imie,nazwisko) {
        super(imie,nazwisko);
        this.nr=nr;
    } 
}

var dyrektor = new Pracownik(1,"Jan","Kowalski");
dyrektor.przedstawSie();

for (wlasnosc in dyrektor) { console.log(wlasnosc);}
```

Wyjście:

```
Jan Kowalski
imie
```

```
nazwisko
nr
```

Na co należy zwrócić uwagę:

- Klasa jest widoczna dopiero od chwili deklaracji (miejsce w programie, gdzie jej nie widać nazywa się czasem "tymczasową strefą martwą", a widoczność przed deklaracją - jak w przypadku funkcji - hoistingiem);
- W kodzie klasy obowiązuje "tryb ścisły" ("strict"). Czyli sprawdzane jest restrycyjnie, czy użyte zmienne są zadeklarowane, czy nie próbujesz użyć dwókrotnie takiej samej nazwy parametru funkcji, lub nazwy klasy do nazwania funkcji / metody etc...
- Wywołanie konstruktora klasy inaczej niż w instrukcji tworzenia obiektu (new Konstruktor()) powoduje zgłoszenie błędu;
- Można wyliczać nazwy własności instrukcji for/in, ale nie dotyczy to metod.

Symbole

W ES6 wprowadzono typ który działa podobnie jak symbole true i false. One nie są używane w obliczeniach, ale wyłącznie do sprawdzenia, czy jakaś zmienna ma taką wartość (w TypeScript wprowadzono dodatkowo typ wyliczeniowy, który ma taką własność). Przykład:

```
> let symbol=Symbol();
undefined
> let symbol2=Symbol('s2');
undefined
> String(symbol2);
'Symbol(s2)'
> String(symbol);
'Symbol()'
>
```

Rozpakowanie struktury

W niektórych językach programowania (np. Python) istnieje możliwość potraktowania kolekcji (struktury) danych jako zbioru odrębnych zmiennych. ES6 także wprowadza taką funkcjonalność:

```
let osoby = [
    { imie : "Jan", nazwisko : "Kowalski"},
    { imie : "Józef", nazwisko : "Nowak"}
];
```

```
for ({ imie, nazwisko } of osoby) {
    console.log(imie+" "+nazwisko);
}
```

Możliwości rozpakowania w Javascript mogą być wykorzystane także do struktur zagnieżdżonych.

Operator rozpakowania i parametr reszty

Dodatkową możliwością wprowadzoną w ES6 są parametry reszty (rest parameter). Jest to parametr krujący listę dowolnej ilości parametrów.

Na przykład:

```
function sumuj(czynnik, ...reszta) {
  for (liczba of reszta) czynnik+=liczba;
  return czynnik;
}

console.log(sumuj(2,4,6,9));
```

Po trzech kropkach podajemy identyfikator pod którym kryje się lista pozostałych parametrów.

Podobna składnia jest używana dla operatora rozpakowania (spread). Stosuje się go tam gdzie mamy wyliczyć wszystkie elementy listy lub struktury.

Przykład:

```
> let lista = ['b', 'c'];
> console.log(['a',...lista,'d','e'].concat());
[ 'a', 'b', 'c', 'd', 'e' ]
```

Podobnie możemy zrobić ze strukturami:

```
> let struktura = {2:'b', 3:'c'};
> let struktura2={1:'a', ...struktura,4:'d'};
> for (i in struktura2) {
.....console.log(i.toString()+' => '+struktura2[i]);
...}
1 => a
2 => b
3 => c
4 => d
```

Iteratory i generatory

Pętla for / of przebiega przez wszystkie elementy listy. Zamiast listy może być dowolny iterator - czyli specjalny obiekt z wbudowaną funkcją next. Generator to funkcja tworząca iteratory. Oznacza się go gwiazdką po słowie function. Kolejne elementy iteracji zwracane są instrukcją yield https://developer.mozilla.org/pl/docs/Web/JavaScript/Referencje/Operatory/yield:

Przykład:

```
> function* generator() {
      for (c=0; c<=9; c++) yield c;
 }
> var iterator_cyfr=generator();
> var suma=0;
> for (c of iterator_cyfr) suma+=c;
45
```

Najważniejsze jest to, że ciało funkcji generatora (tu: `generator()`) nie jest wykonywane w chwili utworzenia iteratora (tu: `var iterator_cyfr=generator();`), ale dopiero gdy wywołamy metodę `next()` iteratora. W powyższym przykładzie dzieje się to (niejawnie) w kolejnych iteracjach pętli for/of. Słowo kluczowe **yield** oznacza, że w dane po nim następujące są zwracane do iteratora, a dalsza część ciała funkcji jest wykonywana dopiero przy następnym wywołaniu `next()` .

Więcej informacji na temat generatorów: http://2ality.com/2015/03/es6-generators.html

Moduły

Mpduły to wydzielone w odrębnych plikach fragmenty kodu Javascript. Ich zawartość może być dołączana poleceniem import. Nie są jednak dołączane wszystkie zaimplementowane w module zmienne i funkcje, ale jedynie te, które zostaną jawnie wyeksportowane i zaimportowane.

Przykład:

Moduł m1.js

```
const klucz=9;

export function szyfr(zmienna) { return zmienna + klucz; }
```

Moduł m2.js

```
import { szyfr } from "./m1.js";

console.log(szyfr(8));
```

Łańcuchy znaków i wyrażenia regularne

ES6 wprowadza dodatkowy typ łańcuchów znaków - w apostrofach odwróconych (` - ang. backtick). Mogą one zawierać oznaczenie miejsc wstawienia danych. Mogą też (w przeciwieństwie do tradycyjnych łańcuchów znaków) obejmować więcej niż jeden wiersz.

Przykład:

```
> const witaj = `Witaj ${name}
   na naszej stronie`;
> let name="Jan Kowalski";
> console.log(witaj);
Witaj Jan Kowalski
na naszej stronie
```

Zamiast prostych zmiennych mogą być wstawiane wyniki złożonych wyrażeń, obiekty, a nawet wzorce podrzędne.

Przykład (ciąg dalszy poprzedniego przykładu):

```
> let page="Nasza strona";
> const h = `Witaj ${name}
  na stronie ${page}
  [${ new Date().toISOString().slice(0, 10) }]`;
> console.log(h);
Witaj Jan Kowalski
na stronie Nasza strona
[2018-05-20]
```

Wprowadzono także możliwość definiowania własnych funkcji przetwarzających takie szablony.
Przykład:

```
> function tagH1(literals, ...substitutions) {
    let result="<h1>";
    substitutions.forEach( (s,i) => {  result+=literals[i]+s; } );
    return result+"</h1>";
  }
> console.log(tagH1`Witaj ${name} na ${page}`);
<h1>Witaj Jan Kowalski na Nasza strona</h1>
```

Objaśnienie:

- funkcja forEach symuluje pętlę - wywołuje ona dla każdego elementu listy funckję (tu: strzałkową) - podaną w parametrze; przekazuje element i jego indeks;
- funkcja tagH1 ma zwrócić przetworzony łańcuch znaków
- zapis j `` `nazwaFunkcji `` literał```oznaca wywołanie funkcji (nazwaFunkcji) z podzielonym na elementy literałem (parametr literals) i elementami do wstawienia (substitutions).

Obietnice i przetwarzanie asynchroniczne

W przeglądarkach wiele czynności odbywa się asynchronicznie. Na przykład program "ściąga" dane w tle - w czasie, gdy użytkownik przegląda stronę. W Javascript tego typu asynchroniczne przetwarzanie jest obsługiwane przez funkcje powrotu (callback), wywoływane po skończeniu asynchronicznie wykonywanej czynności. ES6 wproadza jednak nowe rozwiązanie o nazwie Promise (obietnica): https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

Przykład:

```
console.log('Start...',Date.now());
const delay=5000;
(new Promise(
        (resolve, reject) => {  setTimeout( resolve,  delay ); }
    )
).then(
          () => { console.log('KONIEC',Date.now()); }
      );
```

Objaśnienie:

- Tworzony jest obiekt obietnicy (new Promise), którego zainicjowanie polega na ustawieniu opóźnienia (setTimeout) o 5000 milisekund (zmienna delay).
- funkcja then wywoływana na rzecz tego obiektu implementuje działania podjęte wówczas, gdy obietnica zostanie spełniona (minie 5000 milisekund)

Obiektnica może znajdować się w jednym z następujących stanów:

- spełniona/rozwiązana [*fulfilled/resolved]* - zadanie zrealizowane i zwrócona prawidłowa wartość
- odrzucona [*rejected*] – zadanie nie zrealizowane (wyjątek, nieprawidłowa wartość etc...);
- oczekująca [*pending*] – utworzono obietnicę, ale jeszcze nie ma wyniku;

W powyższym prostym przykładzie nie uwzględniono sytuacji odrzucenia. Obietnica wywołuje wówczas funkcję reject. Szczegółowe objaśnienie: http://devenv.pl/obietnice-promises-podstawy-jezyka-javascript/

Inne

ES6 wprowada także szereg pomniejszych zmian, jak rozszerzenie funkcjonalności niektórych obiektów (z najważniejszych: klasa Number ma teraz własność isInteger), nowe typy danych (zbiory Set) czy dostęp do funkcji interpretera ("silnika") przez tak zwane proxy.

Typescript

Do testowania przykładów Typescript możemy użyć konsoli ts-node (https://www.npmjs.com/package/ts-node\. Instalujemy ją poleceniami:

```
npm install -g ts-node
npm install -g typescript
```

Uruchomienie:

```
ts-node --type-check
```

Typy danych

TypeScript wymaga jawnego podania typu danych w deklaracji zmiennych. Po nazwie zmiennej stawiamy dwukropek (`:`) i podajemy nazwę typu.

Przykłady.

```
let warunek : boolean = true;
```

```
let komunikat: string = 'Możesz użyć apostrofów';
let komunikat2: string = "... albo cuzysłowów";
```

Można też tworzyć wzorce pozwalajace na składanie napisów z elementów:

```
let wynik: string = `OK`;
let komunikat: string = `Wynik działania: ${wynik}.`
```

Liczby mogą być ałkowite, zmiennopzecinkowe, szesnastkowe, binarne i ósemkowe:

```
let integer: number = 38;
let decimal: number = 45.67;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
```

Tablice

Typy tablic można zapisać na dwa różne sposoby:

```
let litery: string[] = ['A', 'B', 'C'];
```

lub jako typ tablicowy (znany z języka Java):

```
let litery: Array<string> = ['A', 'B', 'C'];
```

Krotki

Są to tablice z określoną liczbą elementów określonych z góry typów (słowo "krotka" pochodzi z teorii baz danych i odnosi się do wartości rekordu / wiersza tablicy).

```
let krotka: [number, string] = [1, 'jedynka'];
```

Proszę pamiętać, że kolejność elementów jest tu ważna! .

Wyliczeniowe

Typy wyliczeniowe to sposób na wprowadzenie symboli (stałych) zgromadzonych pod jednym identyfikatorem (nazwą typu wyliczeniowego). Odwołanie do wartości z użyciem znanej notacji kropkowej. Pod symbolami w definicji typu kryją się kolejne numery. Domyślnie wyliczenia zaczynają się od cyfry 0. Można to zmienić prez podstawienie - na przykład `= 7` :

```
> enum Numerki{ zero, jeden, siedem = 7, osiem };
> Numerki.jeden;
1
> Numerki.osiem;
8
```

Inne

- any - dowolny typ,
- void - nieokreślony (np. funkcja nie zwracająą wyniku),
- never - nigdy (na przykład nieskończona pętla)

```
function infiniteLoop(): never {
    while (true) {
```

```
    }
    return 'to się nigdy nie wykona';
}
```

Typy parametrów i wyników funkcji

Do argumentów można dodać typy, a także typ zwracanej funkcji:

```
function iloraz(l: number, m: number): number {
   if (m==0) { return null; }
   else {  return l / m;}
}
```

Słowo kluczowe type

Słowo kluczowe `type` definiuje alias do typu.

```
> type liczba = number;
> type LiczbaLubNumer = string | liczba;
> let n : liczba;
> let ln : LiczbaLubNumer;
> n=1
> ln=1
> ln='abc';
'abc'
> n='abc';
Thrown: × Unable to compile TypeScript
[eval].ts (1,1): Type '"abc"' is not assignable to type 'number'. (2322)
```

KLASY

Własności publiczne, prywatne, chronione

Do klas wprowadzonych przez ES6 TypeScript dodaje możliwość dzielenia własności na publiczne, prywatne i chronione:

- Publiczny (**public**) - swobodny dostępny spoza klasy/obiektu (domyślnie).
- Prywatne (**private**) - dostępne tylko w metodach implementowanych wewnątrz klasy.
- Chronione (**protected**) - podobne do prywatnych, jednak dostępne z klas pochodnych

Przykład:

```
class Szyfr0 {
  private tajny: number;
  protected klucz: number;

  constructor(klucz: number) {
    this.klucz = klucz;
    this.tajny = 123;
  }
  szyfruj(x : number): number { return x + this.klucz + this.tajny; }
  deszyfruj(x : number): number { return x - this.klucz - this.tajny; }
}

class Szyfr1 extends Szyfr0 {
  szyfruj(x : number): number { return super.szyfruj(x) - this.klucz }
  deszyfruj(x : number): number { return super.deszyfruj(x) + this.klucz }
}

var koder = new Szyfr1(44);
alert(koder.szyfruj(9));
```

Zwróć uwagę na przedrostek super - powalający odwołać się do pól klasy nadrzędnej.

Właściwości statyczne

TypeScript pozwala również na definiowanie elementów `statycznych` na klasie. Oznacza to, że możesz użyć właściwości i metod, nawet jeśli nie tworzysz instancji klasy (czyli obiektu).

```
class Stala {
    static przedrostek: number = 48;
}
console.log(Stala.przedrostek);
```

Interfejsy

Interfejs jest opisem wymagań wobec definicji klasy/obiektu. Nie zawiera implementacji.

Przykład:

```
> interface Figura {
   rodzaj : string;
   wymiary : Array<number>;
 }
> class Kwadrat implements Figura { rodzaj = 'kwadrat'; wymiary=[]; }
> var k = new Kwadrat;
> console.log(k.rodzaj);
kwadrat
```

Dekoratory

Typescript wyprzedził rozwój języka Javascript, wprowadzając dekoratory własności, klas i parametrów.

Przykład wyjaśniający działanie dekoratorów:

```
function dekorator1(target: Object, propertyKey: string,
                    descriptor: TypedPropertyDescriptor<any>) {
    let originalMethod = descriptor.value;
    descriptor.value = function(...args: any[]) {
            console.log('Wywołanie z parametrami:');
            console.log(args);
            return originalMethod.apply(this, args);
    };

    return descriptor;
}


class MyClass {

  @dekorator1
  public test(something : string) : string {
    return 'test z parametrem: ' + something;
  }

}

var o= new MyClass();
console.log(o.test('abc'));
```

Funkcja dekorator definiuje dekorator, którym następnie "dekorujemy" metodę "test" klasy MyClass. Taki mechanizm powoduje, że gdy użyjemy metody test(), zostanie wywołana funkcji podana jako wartość descriptor.value. Ona może wykonać jakieś dodatkowe operacje (tu - wypisuje paametry) i wtedy uruchomić właściwą funkcję. Najczęściej używa się tego mechanizmu do ustawienia domyślnych parametrów lub parametrów środowiska w którym funkcja działa.

Typy generyczne

W Typescript zaimplementowano znane z języka Java typy generyczne (generics) - czyli możliwość definiowania zmiennych i funkcji dla typów nieokreślonych ściśle. Można używać kilku typów, bez konieczności ich konwersji (rzutowania). Przy deklaracji i odwołaniu do takiego

typu używamy nawiasów większy/mniejszy: `<>` .

```
> class C<T> {
   wlasnosc : T;
   public ustaw(w : T) { this.wlasnosc=w;}
   public czytaj():T {return this.wlasnosc}
 }
> let o1=new C<number>();
> o1.ustaw(1);
> o1.czytaj();
1
> let o2=new C<string>();
> o2.ustaw(1);
Thrown: × Unable to compile TypeScript
[eval].ts (1,10): Argument of type '1' is not assignable to parameter of type 'string
> o2.ustaw('1');
> o2.czytaj();
'1'
```

Możemy ograniczyć typy, jakie mogą być użyte - na przykład poprzez zastosowanie interfejsów. Służy do tego słowo kluczowe extends (jego znaczenie jest w tym miejscu nieco inne niż w przypadku definiowania klas). Na przykład (https://www.typescriptlang.org/docs/handbook/generics.html):

```
interface Lengthwise {
    length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
    console.log(arg.length);  // Now we know it has a .length property, so no more er
    return arg;
}
loggingIdentity({length: 10, value: 3});
```

Powyżej zawarto tylko najważniejsze elementy Typescript, rozszerzające standard ES6. Dodatkowo można polecić na przykład opis: https://typeofweb.com/2016/07/11/typescript-czesc-1/

Bloki i zakres widoczności zmiennych

Blokiem kodu nazywamy ciąg instrukcji zawartych w nawiasach klamrowych `{}`. Może to być ciało funkcji albo blok wykonywany w instrukcjach złożonych. Jedną z najważniejszych zasad funkcjonowania bloków, jest widoczność zadeklarowanych w nich zmiennych.

Przykład:

```
> if (true) {
   var v=1;
   const c=1;
   let l=1;
  }
> v
1
> l
ReferenceError: l is not defined
> c
ReferenceError: c is not defined
>
```

Widzimy, że zmienna zadeklarowana słowem **var** jest widoczna poza blokiem w którym to wykonaliśmy (caiło instrukcji if), gdy tymczasem w przypadku let i const - mamy widoczność jedynie w obrębie ciała instrukcji/funkcji. W Javascript operuje się często pojęciem domknięcia (clousre) - które określa w jakim bloku jest zamknięta nasza zmienna. W przypadku zmiennych **var** domknięcie tworzą jedynie funkcje. Warto przy tym zwrócić uwagę na niebezpieczeństwo związane z zagnieżdżaniem bloków. Przeanalizujmy poniższy kod:

```
> var kolor1='red';
> var kolor2='blue';
> function komunikat(msg) {
   var kolor1='yellow';
   kolor2='black';
   console.log(msg,kolor1,kolor2);
 }
> console.log(kolor1,kolor2);
red blue
> komunikat('OK');
yellow black
> console.log(kolor1,kolor2);
red black
```

Wykonanie funkcji `komunikat` spowodowało zmianę wcześniej zadeklarowanej zmiennej kolor2, gdy tymczasem kolor1 zmienił się jedynie w ciele funkcji. Różnica bierze się z tego, że w ciele funkcji zadeklarowano zmienną lokalną kolor1 (użycie słowa kluczowego var). Tymczasem zmienna kolor2 jest zadeklarowana tylko raz i ona jest uzywana zarówno w ciele funkcji jak i na zewnątrz (jest to ta sama zmienna).

Problem w tym, że deklaracja zmiennej w Javascript następuje przy pierwszym jej użyciu i brak var w przypadku `kolor2='black';` mógł wynikać z omyłkowego ominięcia var!

Tego problemu nie ma w trybie ścisłym ("strict") skryptów. Wymagana jest wówczas pełna deklaracja zmiennych.

Poniższy skrypt zakończy się błędem - gdyż w pierwszej linii włączamy tryb ścisły:

```
"use strict";
a=1; // brakuje słowa var let lub const
```

Hoisting

Przyjrzyjmy się przykładowi:

```
function komunikat(msg) {
  if (kolor===undefined) {
     var kolor='black';
  }
  console.log(msg,kolor);
}
komunikat('OK');
```

W warunku instrukcji **if** sprawdzamy, czy zmienna kolor ma wartość niezdefiniowną. Jednak ta zmienna nie została jeszcze w ogóle zadeklarowana - dzieje się to w wierszu następnym. Czyli mamy do czynienia z błędem? Nie - gdyż zmienne zadeklarowane z użyciem słowa kluczowego **var** są widoczne w obrębie całej funkcji, w której znajduje się deklaracja (jeśli poza funkcją - deklarujemy tak zmienne globalne). Nazywa się to hoistingiem (http://poradnik.drogimex.pl/2017/05/20/hoisting-zmiennych-i-funkcji-w-javascript/, https://www.nafrontendzie.pl/zakres-zmiennych-javascript).

Nie dotyczy to zmiennych deklarowanych przy użyciu **let** i **const**. One są widoczne dopiero po zadeklarowaniu. Zmiana var w powyższym przykładzie na let/const powoduje błąd.

Notacja kropkowa

[w przygotowaniu]

Deklaratywny styl programowania

Do momentu pojawienia się biblioteki AngularJS programowanie w JavaScript miało charakter imperatywny. Czyli programista opisywał szczegółowo – jak i co program ma wykonać. Jquery ułatwiał jedynie operowanie na drzewie DOM. Wzorowany na tej bibliotece AngularJS wprowadził możliwość definiowania sposobu działania poprzez własne znaczniki HTML (dyrektywy). Był to początek trendu rozwoju programowania deklaratywnego. Opisujemy, co ma być zrobione, a nie jak (to zawiera biblioteka). Szczegółowy opis w Javascript ma dotyczyć algorytmu (logiki biznesowej), a nie jego implementacji w środowisku przeglądarki.

Angular 1.x

Zobaczmy najprostszy przykład programu z użyciem Angular 1.

```
<html ng-app="app">

<head>
  <meta charset="utf-8">
  <script type="text/javascript" src="https://code.angularjs.org/1.6.7/angular.js"></
</head>

<body>
  <div ng-controller="MyController">
    <p>Wiaj {{ name }}!</p>
  </div>
</body>

</html>

<script type="text/javascript">
  (function () {
    var app = angular.module("app", []);
    var MyController = function ($scope) {
      $scope.name = "Jan Kowalski";
    }
    app.controller("MyController", ["$scope", MyController]);
  }());
</script>
```

Widzimy, że pojawiły się nowe własności znacznikiów - dyrektywy ng-app i ng-controller. Są one wykorzystywane do opisu tego, jaki efekt chcemy osiągnąć. Dodatkowo używane są szablony - takie jak dotąd były stosowane przy renderowaniu stron na serwerze. Zapis: `{{ name }}` wskazuje miejsce w którym należy wstawić dane. Aplikacja Angulara to moduł (angular.module) zawierający funkcję kontrolera (`function($scope)`), w której podajemy dane do renderowania (wypełnienia szablonu).

Zapis ten można nieco skrócić:

```
<html ng-app="app">

<head>
  <meta charset="utf-8">
  <script type="text/javascript" src="https://code.angularjs.org/1.6.7/angular.js"></
</head>

<body>
  <div ng-controller="MyController2">
    <p>Wiaj {{ name }}!</p>
  </div>

</body>

</html>

<script type="text/javascript">
    (function () {
      angular.module("app", []).controller("MyController2",
        [ "$scope",
          function ($scope) {
           $scope.name = "Józef";
          }
        ]
      );
    }());
</script>
```

Kolejny przykład pokazuje użycie dyrektywyng-repeat – dla listy elementów, oraz ng-click. Zapis `ng-repeat='item in list_items'` skazuje, że item przyjmuje kolejne wartości z listy items.

```
<html>
<head>
  <meta charset="utf-8">
  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js">
  </script>
</head>
```

```
<body>
<div ng-app="myApp"  ng-controller='MyController3'>
 <ul>
    <li ng-repeat='item in list_items'>{{item}}</li>
 </ul>
 <p>licznik={{ count }}</p>
 <button ng-click='addListItem()'>Dodaj element</button>
  <button ng-click="myFunction()">Zwieksz licznik!</button>
</div>
</body>
</html>

<script>
// https://www.w3schools.com/angular/angular_events.asp
  var app = angular.module("myApp", []);
  app.controller('MyController3', function ($scope) {
    $scope.list_items = [
      'Element 1',
      'Element 2',
      'Element 3'
    ];
    $scope.count = 0;
    $scope.addListItem = function () {
      $scope.list_items.push('Element 4');
    };
    $scope.myFunction = function() {
        $scope.count++;
    }
  });
</script>
```

Pewną wadą Angulara jest to, że jeśli z jakichś powodów nastąpi błąd Javascript (nie wczyta się biblioteka?) - na stronie zobaczymy szablon - z zaznaczonymi polami (`{{ }}`). Jednak ta idea programowania została zastosowana także w rozwiązaniach pozbawionych tej wady (będzie o nich mowa w dalszej części podręcznika).

Poznajemy React – na prostym przyładzie.

Tworzymy aplikację

Stwórzmy aplikację (frontend) – wykorzystując projekt Create React App
https://github.com/facebookincubator/create-react-app

Nazwijmy naszą aplikację "XO":

```
npx create-react-app xo
```

```
├─ y18n@3.2.1
├─ yallist@2.1.2
├─ yargs-parser@5.0.0
└─ yargs@7.1.0
Done in 115.14s.

Success! Created xo at /media/jurek/dev/elearning/react/xo
Inside that directory, you can run several commands:

  yarn start
    Starts the development server.

  yarn build
    Bundles the app into static files for production.

  yarn test
    Starts the test runner.

  yarn eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can’t go back!

We suggest that you begin by typing:

  cd xo
  yarn start

Happy hacking!
jurek@transmem:/elearning/react$
```

Wykonujemy posłusznie:

```
cd xo
yarn start
```

Po chwili mamy komunikat:

```
Compiled successfully!

You can now view xo in the browser.

  Local:            http://localhost:3000/
  On Your Network:  http://192.168.100.101:3000/

Note that the development build is not optimized.
To create a production build, use yarn build.
```

Równocześnie przeglądarka otwiera się na stronie http://localhost:3000/ i widzimy efekt naszej pracy:

To get started, edit `src/App.js` and save to reload.

Co tam zostało zrobione?

1) Plik HTML: **public/index.html** z znacznikiem root, który wskazuje miejsce wstawienia aplikacji:

```
<div id="root"></div>
```

2) Wstawieniem aplikacji (modyfikacją DOM) zajmuje się napisany w React skrypt **src/index.js**

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import registerServiceWorker from './registerServiceWorker';
```

```
ReactDOM.render(<App />, document.getElementById('root'));
registerServiceWorker();
```

3) Sama aplikacja jest w pliku src/App.js a jej wygląd opisano w src/App.css.

React jako generator HTML

Tworząc własną aplikację zacznijmy od modyfikacji tego pliku (App.js). Będziemy tworzyć aplikację do gry w kółko i krzyżyk.

Zatem potrzebujemy tabeli 3x3:

```
import React, { Component } from 'react';
import './App.css';


class App extends Component {
render() {
  return (
  <table>
  <tr><td></td><td></td><td></td></tr>
  <tr><td></td><td></td><td></td></tr>
  <tr><td></td><td></td><td></td></tr>
  </table>
  );
 }
}


export default App;
```

Dodajmy w css (src/App.css):

```
td{
  width: 60px;
  height:60px;
  border: 1px solid green;
}
```

Mimo, że działa - debugger nakazuje dodanie `<tbody>`

Mamy pierwszą korzyść w stosunku do prostego stosowania HTML: pilnowanie składni.

Uwaga:
W programach przykładowych często stosowane są konstrukcje wprowadzone do JavaScript w ES2015. Na przykład możemy uprościć konstrukcję App, stosując zapis funkcji () =>:

```
const App = () => {
return (
…..
```

Ma to szczególne znaczenie w prostych funkcjach typu lambda (zob. opis Redux na końcu rozdziału).

Własności

Obiekt React (Component) zawiera pole (własność) props, w którym przechowuje własności definiowanego znacznika. Pokażemu dodawanie własności o nazwie „klasa”.

App.js:

```
const App = (props) => {
return (
<table className={props.klasa}>
….
```

index.js:

```
ReactDOM.render(<App klasa="plansza"/>, document.getElementById('root'))
```

Widzimy, że dodając w index.js własność "klasa" możemy użyć jej odwołując się poprzez: **props.klasa.**Zwróćmy uwagę, że użyto identyfikatora className a nie class. Możemy wstawić class (i to nawet zadziała), ale debugger nakazuje zmianę class na className. Nie piszemy HTML'a ale opisujemy jak go wygenerować! React powiela własności html, ale zmienia ich nazwy.

Najistotniejsze spośród zmian identyfikatorów wprowadzonych przez React:

1. Użycie className zamiast class ("class" jest zarezerwowane w JavaScript).
2. ??
3. ??

Popawność definicji możemy sprawdzić zmieniając CSS (App.css):

```
.plansza {
```

```
  background-color: yellow;
  border-spacing: 0px;
  border-collapse: separate;
}
```

Własne znaczniki

Zapis App analogicznie jak znaczników HTML nie jest przypadkiem. Wszystkie obiekty React można traktować jak rozszerzenie struktury strony (własne drzewo takie jak DOM).
Stwórzmy zatem klasę XO (nazwa musi zaczynać się dużą literą) i zamieniamy znaczniki <td> na XO.

Plik **xo.js:**

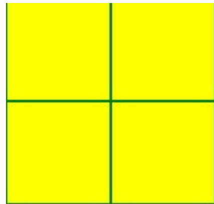
```
import React from 'react';
class XO extends React.Component {
  render() {
    return <td></td>
  }
}

export default XO;
```

Plik **App.js**:

```
import XO from './xo.js';
….
<tr>
<XO></XO>
…
```

Po zainstalowaniu odpowiedniego dodatku do Google Chrome, możemy dokonać „inspekcji” tworzonej strony:



Console Memory Elements Audits Network Sources Performance Application Security React

Highlight Updates Highlight Search

```
▼<App klasa="plansza">
  ▼<table className="plansza">
    ▼<tbody>
      ▼<tr>
        ▼<XO> == $r
            <td></td>
          </XO>
        ▶<XO>…</XO>
```

Dynamika zmian - state

Własności (props) nie są zmieniane inaczej niż w trakcie renderowania strony. Zmiany (ponowne renderowanie) jest inicjowane poprzez zmianę stanu - pola **state**. Wartości z tego pola mogą być użyte w renderowaniu.

Dodajemy zatem **state**, zakładając, że zmiany nastąpią po kliknięciu (zdarzenie onClick):

```
import React from 'react';

class XO extends React.Component {

  constructor(props) {
    super(props);
    this.state = {
      content: '?'
    };
  }

  onClick() {
    console.log('zmiana');
    alert('tu będzie zmiana');
  }

  render() {
    return <td onClick={this.onClick}>{this.state.content}</td>
  }
}

export default XO;
```

Inicjujemy stan (state) w konstruktorze - wstawiając jedno pole **content**.

Widzimy, że zastosowano tu konwencję znaną z różnych implementacji szablonów HTML. Kod JSX jest zapisywany w generowanej treści w nawiasach klamrowych {}.

Implementujemy funkcję „zmiana” - która ma zmieniać stan naszej aplikacji.

1) Definicja:

```
zmiana(xo) {
  if (xo==' ') return 'x';
  else if (xo=='x') return 'o';
  else return ' ';
}
```

2) Użycie: w miejsce „alert” - zdefiniowanie funkcji zmiany z wykorzystaniem zdefiniowanej funkcji..

```
this.setState( // spowoduje ponowny render()
  (prevState, props) => ({
    content: this.zmiana(prevState.content)
}));
```

Warto zwrócić uwagę na to, że w miejsce parametru podajemy funkcję, która zwraca nowy stan. To częsta konstrukcja w React (i generalnie w JavaScript) - użycie definicji funkcji tam gdzie w tradycyjnym programowaniu pojawia się odwołanie do obiektu.

3) Powiązanie funkcji onClick z kontekstem jej wykonania. Po modyfikacjach w funkcji odwołujemy się do zmiennej obiektowej **this**. To wymaga przekazania (bind) zmiennej dla właściwego obiektu. Możemy we własności – ale lepiej w konstruktorze. Obiektu (dodajemy na końcu konstruktora)

```
this.onClick = this.onClick.bind(this);
```

Bez tego TypeError: Cannot read property 'setState' of undefined

Trochę to dziwne, że konstruktor nie wykonuje tego domyślnie, ale to chyba zaszłość historyczna (można definiować wykonywane funkcje inaczej niż jako metodę tego obiektu).

Uwaga!

Zmieniamy tylko state a nie props. Jeśli w wartości właściwości abc ustawimy {this.state.element}, to props.abc będzie miało wartość taką jak this.state.element.

Jak to opublikować?

Przerywamy yarn i wykonujemy `yarn build` – w katalogu build otrzymujemy naszą stronę.

Ponowne uruchomienie serwera deweloperrskiego (localhost:3000):

```
yarn run start
```

Dalsza część aplikacji zostanie opisana w rozdziale dotyczącym Redux'a.

React-native

Na bazie technologii React powstał projekt React-native, który umożliwia programowanie cross-platform. Czyli można tworzyć aplikacje "natywne" (uruchamiane nie w przeglądarce, ale w samym urządzeniu). Więcej informacji: http://maciejgos.com/programowanie-cross-platform-w-react-native/

ANGULAR

Tworzymy aplikację

Do zainstalowania Typescript i Angulara potrzebujesz środowiska NodeJS.

Instalacja:
`npm install -g @angular/cli`

Po utworzeniu wpisz w linii komend:

```
ng new nazwa_aplikacji
```

Może to zająć kilka chwil. Po utworzeniu, przejdź do `nazwa_aplikacji` i uruchom w terminalu:
`ng serve --open`

Zbudowanie aplikacji w wersji dystrybucyjnej:

```
ng build
```

Tworzy on aplikację w czystym Javascript w katalogu `dist/nazwa_aplikacji/` . Zwróć uwagę na generowany w pliku index.html wpis: `<base href="/">` . Zakłada on, że strona będzie uruchamiana z katalogu głównego (/).:

```
<html lang="en">
<head>
<meta charset="utf-8">
<base href="/">
</head>
<body>
<app-root></app-root>
<script>
// tu jest umieszczany wygenerowany kod
</script>
</body>
</html>
```

`<app-root>` jest komponentem aplikacji, który jest wywoływany, i to jest ten komponent, który wyrenderuje na ekranie. Sekcja `<script></script>` tworzona jest dopiero w czasie budowania aplikacji - nie ma jej we wzorcu strony.

Budujemy komponent

W katalogu src/app zbudujmy komponent na wzór wygenerowanego app.component - powielając przykład planszy do gry w kółko i krzyżyk. Nazwijmy ten komponent xo.component. W konsekwencji implementację zapisujemy w pliku src/app/xo.component.tx:

```
import { Component } from '@angular/core';

@Component({
  selector: 'xo-selector',
  template : `<table className="plansza">
   <tbody>
     <tr>
      <td x="0" y="0"> {{ xo[0] }} </td>
      <td x="1" y="0"> {{ xo[1] }} </td>
      <td x="2" y="0"> {{ xo[2] }} </td>
     </tr>
     <tr>
      <td x="0" y="1"> {{ xo[3] }} </td>
      <td x="1" y="1"> {{ xo[4] }} </td>
      <td x="2" y="1"> {{ xo[5] }} </td>
     </tr>
     <tr>
      <td x="0" y="2"> {{ xo[6] }} </td>
      <td x="1" y="2"> {{ xo[7] }} </td>
      <td x="2" y="2"> {{ xo[8] }} </td>
     </tr>
     <tr>
      <td colSpan="3">{{ stanGry }} </td>
     </tr>
   </tbody>
</table>
`,
  styleUrls: ['./xo.component.css']
})
export class XoComponent {
  stanGry='KÓŁKO I KRYŻYK';
  xo = [' ',' ',' ',' ',' ',' ',' ',' ',' '];
}
```

W pliku xo.component.css umieszczamy style dla naszej planszy (można przekopiować z przykładu w React). Zamiast szablonu w postaci własności `template` możemy użyć własności templateUrl wskazującej na zewnętrzny plik xo.template.html:

```
templateUrl: './xo.component.html',
```

Użyliśmy selektora `selector: 'xo-selector'`, aby przy okazji pokazać możliwość tworzenia strony z wielu komponentów. Nasz główny plik wzorca strony uzupełniamy o ww selektor:

```
<body>
  <app-root></app-root>
  <xo-selector></xo-selector>
</body>
```

Ostatnią rzeczą jaką musimy zrobić, to wskazać Angularowi powiązanie między aplikacją na użytym selektorem. Zmieniamy w tym celu definicję modułu w pliku src/app/app.module.ts:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { XoComponent } from './xo.component';

@NgModule({
  declarations: [
    AppComponent,
    XoComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent, XoComponent]
})
export class AppModule { }
```

Jak widać importujemy komponent, a następnie dodajemy go do deklaracji oraz listy komponentów potrzebnych w chwili startu (bootstrap).

Uzyskujemy oczekiwany efekt - planszę 3x3.

Obsługa zdarzeń

Zmieńmy nieco nasz komponent - tak, aby obsłużyć zdarzenie onclick (kliknięcie):

```
export class XoComponent {
  stanGry: string ='KÓŁKO I KRYŻYK';
  public xo: Array<string>;

 constructor() {
  this.xo = [' ',' ',' ',' ',' ',' ',' ',' ',' '];
```

```
 }

 public click(event, n, item) {
  alert('Kliknąłem ['+n+']:' + item);
 }

}
```

Po wywołaniu metody click - wyświetli się komunikat potwierdzający poprawne wywołanie.

Musimy dodać takie wywołanie do każdej komórki naszej planszy:

```
     <tr>
      <td x="0" y="0" (click)="click($event, 0, xo[0])"> {{ xo[0] }} </td>
      <td x="1" y="0" (click)="click($event, 1, xo[1])"> {{ xo[1] }} </td>
      <td x="2" y="0" (click)="click($event, 2, xo[2])"> {{ xo[2] }} </td>
     </tr>
     <tr>
      <td x="0" y="1" (click)="click($event, 3, xo[3])"> {{ xo[3] }} </td>
      <td x="1" y="1" (click)="click($event, 4, xo[4])"> {{ xo[4] }} </td>
      <td x="2" y="1" (click)="click($event, 5, xo[5])"> {{ xo[5] }} </td>
     </tr>
     <tr>
      <td x="0" y="2" (click)="click($event, 6, xo[6])"> {{ xo[6] }} </td>
      <td x="1" y="2" (click)="click($event, 7, xo[7])"> {{ xo[7] }} </td>
      <td x="2" y="2" (click)="click($event, 8, xo[8])"> {{ xo[8] }} </td>
     </tr>
```

To powinno wystarczyć, aby kliknięcie w komórce planszy wywołało komunikat.

Jednak nam chodzi o to, by stan planszy się zmieniał, a nie tylko o śledzenie kliknięć. W Angularze jest to jeszcze prostsze niż w React. Nowa implementacja funkcji click():

```
public click(event, n, item) {
  let newitem : string = ' ';
  if (item==' ') newitem='x'
  else if (item=='x') newitem='o';
  this.xo[n]=newitem;
}
```

Angular jest dużo bardziej rozbudowanym frameworkiem, niż React. Pełny opis wykracza znacznie poza ramy niniejszej publikacji (w następnym rozdziale opisano jeszcze integrację z Redux).

Programowanie funkcyjne

Programowanie funkcyjne (funkcjonalne) jest odmianą programowania deklaratywnego, z wykorzystaniem funkcji. Czyli zamiast szczegółowego opisywaniarozwiązania problemu (algorytmu), skupiamy się na jego jednoznacznym opisie. Na przykład algorytm liczenia długości listy polega na zliczeniu ilości elementów. Zamiast niego możemy się posłużyć definicją: jest to długość końcówki - po odjęciu pierwszego elementu - zwiększona o 1, przy czym długość pustego łańcucha wynosi zero.

```
długość([]) = 0
długość([pierwszy_element,końcówka]) = długość(końcówka)+1
```

Taką definicję nazywamy rekurencyjną (rekurencja jest często wykorzystywana w programowaniu funkcyjnym). Zapis funkcyjny programów ma liczne konsekwencje (https://4programmers.net/In%C5%BCynieria_oprogramowania/Wst%C4%99p_do_programowania_funkcyjnego):

- zamiast tradycyjnego ciągu instrukcji, wykonywanych w ściśle ustalonej kolejności, mamy opis problemu wyrażany za pomocą matematycznych zależności i wyrażeń (funkcji), przy czym nie znamy kolejności, w której będą one wykonywane.
- Instrukcja przypisania staje się bezużyteczna, gdyż nie znamy chwili w której będzie ona wykonana.
- Nie obsługujemy mechanizmów przydzielania i zwalniania pamięci.
- Ponieważ nie musimy sterować kolejnością wykonania, unikamy wielu potencjalnych błędów.

Skoro nie ma instrukcji przypisania, możemy przyjąć, że program zapisany w języku funkcyjnym nie zmiena danych (stanu pamięci), ale opisuje wynik działania. W praktyce (w Javascript) zakłada się, że zmiana stanu dotyczy całego systemu (wyrenderowanej na nowo strony). Po ustaleniu tego stanu - nie ulega on zmianie. Ciągle jednak można zainicjować nowe renderowanie (znów otrzymujemy nowy stan po jednokrotnym wyliczeniu przez program w języku funkcyjnym.

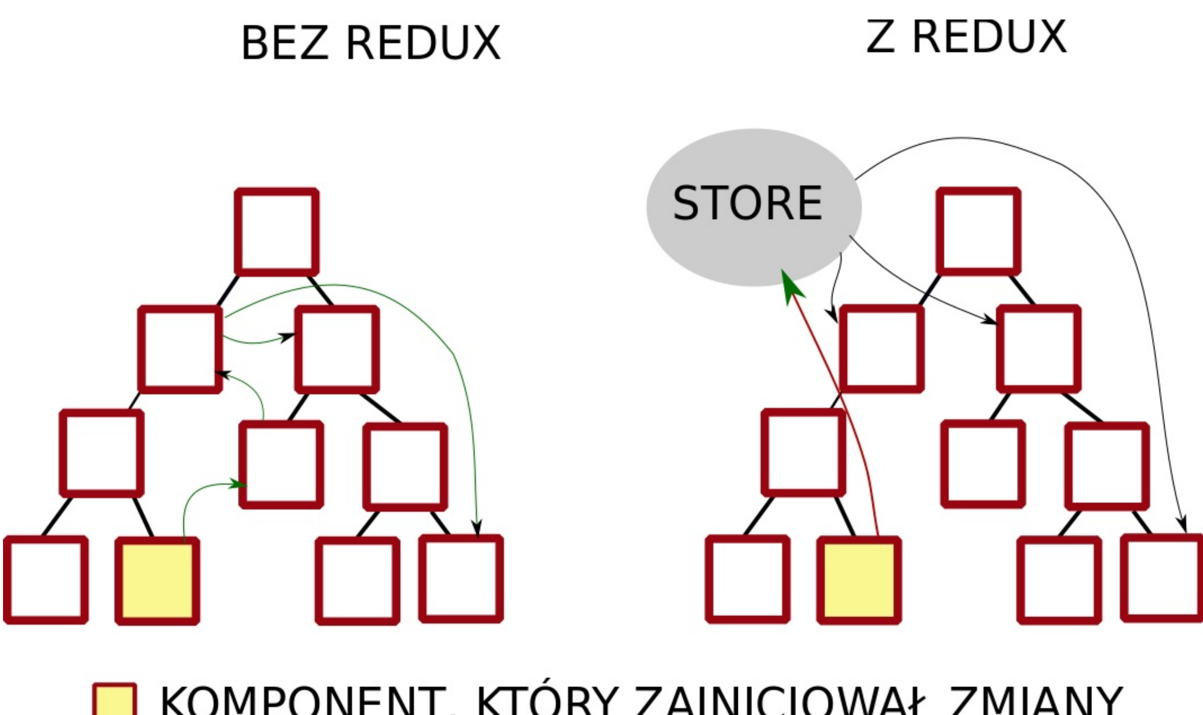
Redux

Wróćmy do naszego przykładu - gry w kółko i krzyżyk w React.

Chcemy powiązać elementy – na przykład zasygnalizować, gdy w naszej grze ktoś wygrał (ma 3 znaki w jednej linii).

Nie wnikając w szczegóły (nie będziemy się tym zajmować) – poza props i state w każdym obiekcie jest context który działa tak jak props, ale udostępnia dane dla potomków. Dzięki temu jest możliwe połączenie między obiektami. Instrukcja bind wspomniana wyżej (this.onClick = this.onClick.bind(this);) umożliwia właśnie zmianę kontekstu przy wywołaniu onClick.

Redux wykorzystuj kontekst implementując pamięć „Storage") - wspólną dla wszystkich obiektów.



Redux nie dostarcza gotowej pamięci (Storage), ale trzeba ją zdefiniować.

Definicja obejmuje:

- store – stan całej aplikacji
- reducer – wyliczający nowy stan
- akcje (dispacher)– czyli funkcje wywołujące pożądane zmiany (reducer dla określonego parametru).

Do tego potrzebujemy trochę „magii”. Poniższa struktura może służyć jako szablon (szkielet) implementacji.

Ponieważ przykład jest stosunkowo prosty - pozostawiono szczegóły implementacyjne (całość: https://github.com/galicea/eduprog/tree/master/web/react/redux/xo).

1) Definiujemy identyfikatory akcji (**src/types/index.js**). Użycie stałych w miejsce nazw (łańcuchów znaków) chroni przed "literówkami":

```
export const ACTION_ZMIANA = 'ACTION_ZMIANA'
```

2) Pamięć (plik src/store.js):

```
import { createStore, } from 'redux';
import * as actionTypes from './types/index';

const StanAplikacji = {
  xo : [], // znak x / o
  kto : '',  // czyj ruch
  stanGry: ''// komunikat
};

function inicjujStanAplikacji() {
  return {
    xo: new Array(9).fill('?'),
    kto: 'x',
    stanGry: '* Kółko i Krzyżyk *'
  };
}

const reducer = (state, action) => {
    if (action.type === actionTypes.ACTION_ZMIANA) {
    let ix = action.y*3 + action.x;
    if (state.xo[ix] !== '?') return state;
    let nxo  = state.xo.slice(); // PŁYTKA KOPIA state.xo.
    nxo[ix] = state.kto;
    let nkto='x';
    let nstan = 'ruch: x';
    if (state.kto==='x') {
      nkto='o';
      nstan = 'ruch: o';
    }
    if (   ((nxo[0] !== '?') && (nxo[0]===nxo[1]) && (nxo[0]===nxo[2]))
         || ((nxo[3] !== '?') && (nxo[3]===nxo[4]) && (nxo[3]===nxo[5]))
         || ((nxo[6] !== '?') && (nxo[6]===nxo[7]) && (nxo[6]===nxo[8]))
         || ((nxo[0] !== '?') && (nxo[0]===nxo[3]) && (nxo[0]===nxo[6]))
         || ((nxo[1] !== '?') && (nxo[1]===nxo[4]) && (nxo[1]===nxo[7]))
         || ((nxo[2] !== '?') && (nxo[2]===nxo[5]) && (nxo[2]===nxo[8]))
         || ((nxo[0] !== '?') && (nxo[0]===nxo[4]) && (nxo[0]===nxo[8]))
         || ((nxo[2] !== '?') && (nxo[2]===nxo[4]) && (nxo[2]===nxo[6]))  ) {
```

```
        nstan = 'KONIEC';}
    return {
     ...state, // spread operator https://redux.js.org/recipes/using-object-spread-op
      xo : nxo,
      kto : nkto,
      stanGry : nstan
    };

  } else {
     return state;
  }
}

export {StanAplikacji};
export const store = createStore(
  reducer,
  inicjujStanAplikacji(),
  // dla debuggera window.__REDUX_DEVTOOLS_EXTENSION__ &&
  // window.__REDUX_DEVTOOLS_EXTENSION__()
);
```

Kluczowe znaczenie ma reducer, który na podstawie stanu i akcji wypracowuje nowy stan. Struktura action zawiera wcześniej zdefiniowany identyfikator i ewentualne parametry (zobacz poniżej). W większych aplikacjach w miejsce **if** stosuje się:

switch (action.type) {

case ACTION_<NAZWA_AKCJI>:

default: return state;

}

Pamięć (store) twozymy funkcją createStore z parametrem reducera i początkową wartością..

Fragment wyliczenia nowego stanu może wygladac następująco:

return Object.assign({ }, state, { <NOWY STAN> });

3) Definiujemy kreatory akcji (plik actions/index.js):

```
import * as actionTypes from '../types/index';

export function zmiana(x,y : string) {
  return {
    type: actionTypes.ACTION_ZMIANA,
    x,
    y
  };
```

```
}
```

4) W aplikacji (App.js) łączymy kontekst z pamięcią Reduxa:

```
import React  from 'react';
import './App.css';
import XO, {Wynik} from './xo.js';
import { connect } from 'react-redux';


class App extends React.Component {

    render() {
        return (
            <table className="plansza">
                <tbody>
                <tr>
                    <XO x={0} y={0} c={this.props.data.xo[0]} />
                    <XO x={1} y={0} c={this.props.data.xo[1]} />
                    <XO x={2} y={0} c={this.props.data.xo[2]} />
                </tr>
                <tr>
                    <XO x={0} y={1} c={this.props.data.xo[3]} />
                    <XO x={1} y={1} c={this.props.data.xo[4]} />
                    <XO x={2} y={1} c={this.props.data.xo[5]} />
                </tr>
                <tr>
                    <XO x={0} y={2} c={this.props.data.xo[6]} />
                    <XO x={1} y={2} c={this.props.data.xo[7]} />
                    <XO x={2} y={2} c={this.props.data.xo[8]} />
                </tr>
                <tr>
                    <td colSpan="3"><Wynik msg={this.props.data.stanGry}/></td>
                </tr>
                </tbody>
            </table> );
    }
}


const mapStateToProps = state => {
    return {
        data: state
    }
};


export default connect(
    mapStateToProps
)(App);
```

Najważniejszą zmianą jest wykonanie"wstrzyknięcia" pamięci do własności obiektów - przy pomocy connect:

import{connect}from'react-redux'; export default connect(...)(App);

Ponieważ mapujemy stan (state) Redux'a na własność o nazwie "data" - możemy się odwoływać do wartości tego stanu poprzez this.props.data....

Jest to coś, co nazywa się "wstrzykiwaniem" funkcjonalności do obiektu.

5). Implementacja XO zmieni się tak, by wywołać zdefiniowaną akcję.

Podobnie jak w przypadku App dodajemy tu funkcjonalność poprzez connect. Ale tym razem odwołanie do akcji.
Funkcja connect jako argumenty przyjmuje dwie funkcje zwyczajowo nazywane mapStateToProps i mapDispatchToProps

mapStateToProps — jako argument przyjmuje cały stan i musi zwrócić propsy dla danego komponentu

mapDispatchToProps — jako argument przyjmuje funkcję lub obiekt z action creators

```
import React  from 'react';
import {connect} from "react-redux";
import {bindActionCreators} from "redux";
import * as Akcje from './actions';

class Wynik extends React.Component {
    render() {
        return <span>{this.props.msg}</span>
    }
}

export {Wynik};

class XO extends React.Component {
    render() {
      return <td onClick={ () =>  this.props.zmiana(this.props.x,this.props.y) }>{thi
    }
}

const mapDispatchToProps = dispatch => bindActionCreators(
    Akcje,
    dispatch,
)

export default connect(
    null,
    mapDispatchToProps
)(XO);
```

W powyższym przykładzie należy dodatkowo zwrócić uwagę na użycie funkcji labda (() => this.....). Taki zapis sprawia, że zmienna obiektowa **this** (jak i ewentualne inne zmienne obiektu) pochodzą z obiektu wywołującego. A więc this zawiera własności z XO. Przy innym zapisie, musielibyśmy użyć **bind(this)**.

6) Ostatnią rzeczą, jaką musimy wykonać jest zmiana głównego renderowania (plik index.js) poprzez dodanie providera dostarczającego pamięć:

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import { store } from './store.js'
import { Provider } from 'react-redux'
import App from './App';

ReactDOM.render(
<Provider store={store}>
<App klasa="plansza"/>
</Provider>,
document.getElementById('root')
);
```

Przed ponownym uruchomieniem (yarn start) wykonujemy:

```
yarn add redux
yarn add react-redux
```

Redux i Anguler

Istnieje kilka implementacji Redux'a dla Angulara (>=2):

- ngrx - implementacja architektury redux - nie jest to obudowa Redux, ale samodzielna implementacja;
- ng2-redux - implementacja połączenia z Redux dla Angular 2
- angular-redux - aktualna implementacja połączenia Reduxa a Angularem >=2

Dalsza część opisu dotyczy angular-redux.

Krótkie wprowadzenie (ang.): https://github.com/angular-redux/store/blob/master/articles/intro-tutorial.md. W języku polskim: https://fsgeek.pl/post/redux-store-w-angularze/

Instalacja

Instalujemy Redux i angular-redux/store (https://github.com/angular-redux/store).

```
npm install --save redux
npm install --save @angular-redux/store
```

Przygotowanie store aplikacji

Korzystamy z poprzedniego przykładu. Zmiany są bardzo niewielkie. Przede wszystkim umieszczamy implementację w plikach *.ts (Typescript). Tworzymy katalog src/app/rx. Tworzymy dwa pliki: actions.ts i store.ts:

W pliku action.ts umieszczamy acje. Powinniśmy je umieścić wewnątrz klasy (tu: xoActions):

```
export class xoActions {
  static ACTION_ZMIANA = 'ACTION_ZMIANA';

  zmiana(x,y : string) {
    return {
      type: xoActions.ACTION_ZMIANA,
      x,
      y
    };
  }
}
```

W pliku store.ts umieszczamy implemnetację pamięci aplikacji (store). Najważniejszą zmianą jest użycie interfejsów do opisu stanu aplikacji:

```
import { createStore,  DeepPartial, Store,  compose  } from 'redux';
import { xoActions } from './actions';

export interface IStanAplikacji { // zamiast export const StanAplikacji = {
    xo : string[], // znak x / o
    kto : string,  // czyj ruch
    stanGry: string// komunikat
  }

export const INITIAL_STATE: IStanAplikacji  = {
    xo: new Array(9).fill('?'),
    kto: 'x',
    stanGry: '* Kółko i Krzyżyk *'
}

export const reducer = (state=INITIAL_STATE, action) => {
    if (action.type === xoActions.ACTION_ZMIANA) {
    let ix = parseInt(action.y,10)*3 + parseInt(action.x,10);
    if (state.xo[ix] !== '?') return state;
    let nxo  = state.xo.slice(); // PŁYTKA KOPIA state.xo.
    nxo[ix] = state.kto;
    let nkto='x';
    let nstan = 'ruch: x';
    if (state.kto==='x') {
      nkto='o';
      nstan = 'ruch: o';
    }
    if (   ((nxo[0] !== '?') && (nxo[0]===nxo[1]) && (nxo[0]===nxo[2]))
         || ((nxo[3] !== '?') && (nxo[3]===nxo[4]) && (nxo[3]===nxo[5]))
         || ((nxo[6] !== '?') && (nxo[6]===nxo[7]) && (nxo[6]===nxo[8]))
         || ((nxo[0] !== '?') && (nxo[0]===nxo[3]) && (nxo[0]===nxo[6]))
         || ((nxo[1] !== '?') && (nxo[1]===nxo[4]) && (nxo[1]===nxo[7]))
         || ((nxo[2] !== '?') && (nxo[2]===nxo[5]) && (nxo[2]===nxo[8]))
         || ((nxo[0] !== '?') && (nxo[0]===nxo[4]) && (nxo[0]===nxo[8]))
         || ((nxo[2] !== '?') && (nxo[2]===nxo[4]) && (nxo[2]===nxo[6]))  ) {
          nstan = 'KONIEC';
    }
    return {
     ...state, // spread operator https://redux.js.org/recipes/using-object-spread-op
      xo : nxo,
      kto : nkto,
      stanGry : nstan
    };

  } else {
     return state;
  }
}
```

```
export const store = createStore(reducer);
```

Definicja modułu powiązanego z Redux

Definicja modułu (plik app.module.ts) musi zostać uzupełniona w polu **providers**, gdzie umieszczamy klasę akcji (tu: xoActions). Z kolei obiekt pamięci "rejestrujemy" poleceniem NgRedux.provideStore (obiekt NgRedux jest parametrem konstruktora):

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { XoComponent } from './xo.component';


// redux
import { NgReduxModule, NgRedux } from '@angular-redux/store';
import { IStanAplikacji, store } from './rx/store';
import { xoActions } from './rx/actions';
//-

@NgModule({
  declarations: [
    AppComponent,
    XoComponent
  ],
  imports: [
    BrowserModule,
    NgReduxModule, // redux
  ],
  providers: [xoActions], // <- redux actions types
  bootstrap: [AppComponent, XoComponent]
})
export class AppModule {

// redux
constructor(ngRedux: NgRedux<IStanAplikacji> ) {
   ngRedux.provideStore(store);
}
//-

}
```

Zamiast rejestrowania gotowego store, możemy go w konstruktorze utworzyć : `ngRedux.configureStore(reducer);`

Użycie store w komponentach

Najprostszym sposobem powiązania store z wyświetlanymi polami komponentów jest przepisanie wartości (w naszym przypadku xo i stanGry). Wtedy wystarczy odpowiednio zdefiniować obsługę zdarzeń i konstruktor . Wykorzystujemy getState().

```
import { Component } from '@angular/core';
// redux
import { NgRedux } from '@angular-redux/store';
import { xoActions } from './rx/actions';
import { IStanAplikacji, store } from "./rx/store";
//-

@Component({
  selector: 'xo-selector',
  templateUrl: './xo.component.html',
  styleUrls: ['./xo.component.css']
})
export class XoComponent {
  stanGry: string ='KÓŁKO I KRYŻYK';
  public xo: Array<string>;

constructor(
  private ngRedux: NgRedux<IStanAplikacji>,
  private actions: xoActions) {
  this.xo = store.getState().xo;
  this.stanGry = store.getState().stanGry;
}

public click(event, n, item) {
  const mapxy = [{x:'0',y:'0'},{x:'1',y:'0'},{x:'2',y:'0'},
                 {x:'0',y:'1'},{x:'1',y:'1'},{x:'2',y:'1'},
                 {x:'0',y:'2'},{x:'1',y:'2'},{x:'2',y:'2'}];
 this.ngRedux.dispatch(this.actions.zmiana(mapxy[n].x,mapxy[n].y));
 this.xo = store.getState().xo;
 this.stanGry = store.getState().stanGry;
}

}
```

To rozwiązanie ma jedną zasadniczą wadę - dane ze store są powielane w polach komponentu. Można ten problem rozwiązać, stosując dekorator własności @select (https://angular-redux.github.io/store/globals.html#select\, https://github.com/angular-redux/store/blob/master/articles/select-pattern.md\)

Na przykład dla pola stanGry możemy zastosować uproszczony zapis:

```
export class XoComponent {
  @select() 'stanGry' ;
  ....
```

We wzorcach zamiast `{{ stanGry }}` użyjemy `{{stanGry | async}}`

TDD

[do uzupełnienia]

https://medium.com/frontend-fun/angular-unit-testing-jasmine-karma-step-by-step-e3376d110ab4

https://stackoverflow.com/questions/47329520/unit-testing-react-using-karma-and-jasmine

Co dalej?

Dobrym repozytorium wiedzy na temat Javascript w środowisku HTML5 jest cykl poświęcony egzaminowi certyfikacyjnemu: https://mrzepinski.pl/tag/javascript

Nieocenionym źródłem bieżących informacji na temat rozwoju jzyka i środowiska Javascript jest portal https://www.nafrontendzie.pl/

Strona : https://github.com/greg-dev/clean-code-javascript-pl to międzynarodowy podręcznik opisujący jak programować w sposób przejrzysty.