PROGRAMISTA JAVA SCRIPT

Kurs zdalny ALX 09.2022

r.wasik@alx.pl

1. HTML5

Podstawą pracy w technologiach Web jest znajomość składni HTML.

Struktura dokumentu HTML5

```
<!DOCTYPE html>
<html>

      <head></head>
      <body></body>

</html>
```

Skrót w VSCode:
Shift+1+enter
html:5+enter

Znaczniki blokowe i znaczniki liniowe (block i inline)

Znaczniki blokowe zajmują całą szerokość strony. Można ustawiać ich wysokość i szerokość. Nadają się do budowania elementów strony.

```
<div></div>
<p></p>
<ol></ol>
<ul></ul>
```

Znaczniki liniowe występują jeden po drugim obok siebie. Nie mają wysokości ani szerokości. Nadają się do formatowania tekstu.

Skrót w VSCode: Kopiowanie elementu
ustaw kursor - Shift+alt+strałka kierunkowa
ustaw kursor - ctrl+C, ctrl + V

```
<i></i>
<b></b>
<img>
```

```
<img src="https://www.stava.pl"> alt="tekst alternatywny">
```

Przy znaczniku mamy atrybuty: src oraz alt.
src, alt – to nazwy atrybutów
w cudzysłowach podajemy wartość atrybutu

```
<sup></sup>        Indeks górny
<sub></sub>        Indeks dolny
<span></span>      konkretny element wewnątrz linii, znacznika liniowego
<br>
```

Klasy, identyfikatory

```
<znacznik class="nazwa-klasy"></znacznik>
<znacznik id="nazwa-identyfikatora"></znacznik>
```

Łącza, tabele

https://datatables.net

```
<table></table>
<tr></tr>          table row
<td></td>          table divide
<th></th>          table heading
<td colspan="2"></td>        jedna kolumna scalona z dwóch kolumn
<td rowspan="2"></td>        jedna kolumna scalona z dwóch wierszy
```

Zawsze wprowadzamy te atrybuty do <td> kolumny, nigdy do <tr> wierszy

Formularze

GUI = Graphic User Interface

```
<form></form>
<form action="#" method="#"></form>
```

atrybut action nas nie interesuje na tym kursie
atrybut method – jak działa

Metoda „get” powoduje, że dane, które wpiszemy w formularz są wysyłane do url. Nadaje się np. do wyszukiwarki. Nie nadaje się do logowania.

Metoda „post” powoduje, że dane, które wpiszemy w formularz są wysyłane do protokołu https, który jest zaszyfrowany. Nie widać tych danych nigdzie, są ukryte. Nadaje się do logowania.

```
<label for=”x”>Login</label>
<input type=”text” id=”x” name=”login”>
```

Etykieta dla elementu o id „x”
Oprócz etykiet można też tworzyć placeholdery wewnątrz inputu. Placehoder to atrybut.

Każde pole formularzowe musi mieć atrybut name. Pozwala rozróżnić on przesyłane dane

Input typu radio jednokrotnego wyboru
Atrybut value określa wartość domyślną tego elementu wysyła tę wartość do url

```
<input type=”radio” name=”kolor” value=”white”>Biały
<input type=”radio” name=”kolor” value=”red”>Czerwony
<input type=”radio” name=”kolor” value=”blue”>Niebieski
```

Jeśli pole ma być jednokrotnego wyboru, to wszystkie muszą mieć jednakowy name, ale różną wartość

Input typu select

```
<select name=”miasto”>
      <option value=”0”>wybierz…</option
      <option value=”Kraków”>>Kraków</option
      <option value=”Warszawa”>>Warszawa</option
      <option value=”Gdańsk”>>Gdańsk</option
</select>
```

Input textarea

```
<textarea name=”uwagi”>tutaj twój text</textarea>
```

Button formularza

```
<button type=”submit”>wyślij</button>
<button type=”reset”>wyczyść</button>

<input type=”submit”>wyślij</button>
<input type=”reset”>wyczyść</button>
```

Automatyczne aktywowanie pola.
Jest to wygodne i stosujmy, żeby ułatwić użytkownikowi.
Służy do tego atrybut autofocus

Pola obowiązkowe i komunikaty walidacji.
Służy do tego atrybut required.

```
<input type=”text” id=”x” name=”login” autofocus required>
```

TE ROZWIĄZANIA NIE SĄ JEDNAK BEZPIECZNE SAME! Ponieważ można html zmanipulować w narzędziach developerskich przeglądarki.
NALEŻY JE ZABEZPIECZYĆ JESZCZE I ZWERYFIKOWAĆ JAVA SCRIPTEM.

```
<progres value=”50” max=”100”></progres>
```

Ten element może być sterowany javascriptem. Pasek postępu.

2. CSS

Składnia pojedynczego stylu:

```
Cecha:wartość;
```

Style lokalne:
To style dołączone do danego znacznika html jako jego atrybut.

Style zagnieżdżone:
To style umieszczone w znaczniku <head> dokumentu html

Style zewnętrzne:
To style umieszczone w odrębnym pliku i podlinkowane do dokumentu html.

Jednostki miary
Jednostki bezwzględne: px
Jednostki względne: % - dobre do budowania elementów strony responsywnej
vw - vieport width – szerokość obszaru roboczego
vh - vieport height – wysokość obszaru roboczego
vmax, vmin – bierze największą wartość wymiarów viewportu (szerokość lub wysokość)

Style tekstu
text-align, line-height, text-decoration, font-family, font-size, font-weight, font-style

Fonty hostowane.
Link do nich umieszczamy w znaczniku <head> dokumentu html
Nazwy dwuczłonowe fontów muszą być ujęte w pojedynczym cudzysłowie (apostrofy).

Kolory
color: red, green, blue
color: #000000
color: rgb(143,243,78)
color: rgba(143,243,78, 0.5)
background-color
background-image:url(..jpg); background-repeat:no-repeat; background-position, background-size:cover

Model pudełkowy
margin, border, padding.;
margin – margines zewnętrzny
padding – margines wewnętrzny – kolejność odsunięć zgodnie ze wskazówkami zegara

Selektory
znacznikowe: znaczniki, #id, .class

Selektory złożone
Złożone z kilku elementów. Są bardziej specyficzne, dokładniej wskazują element.

Hierarchia
Style inline w dokumencie html nadpisują inne style.
Style umieszczone w niższych liniach nadpisują style powyższe.

3. JAVA SCRIPT

W Java Script kod jest czytany i wykonywany od góry do dołu.

Najpierw musimy zdefiniować zmienną, by potem jej użyć. Inaczej wyświetli nam błąd.

W html5:

```
<script>
      var imie=”Agnieszka”;
      alert („Agnieszka”);
</script>
```

Dzięki Java Script możemy tworzyć nie tylko już strony internetowe, ale całe aplikacje internetowe.
Praktycznie współczesna strona www nie obejdzie się bez JavaScript.

Pliki .js linkuje się w sekcji head dokumenty html – tak jak css. – robi się je w osobnym pliku.
Można też oczywiście używać osadzonych skryptów.

Komentarze
Wierszowe i blokowe

```
// komentarz wierszowy
/* komentarz blokowy */
```

Instrukcje
Instrukcje to polecenia do wykonania. Są one interpretowane przez przeglądarkę.
Każdą instrukcję musimy zakończyć znakiem średnika, który oznacza koniec instrukcji.

```
Console.log(„Agnieszka”);
```

W narzędziach developerskich przeglądarki Konsola służy do komunikacji JavaScriptu z tym co jest wyświetlane w okienku konsoli.

Zmienne. var
Zmienna to miejsce w skrypcie, w którym można przechowywać dane, czyli liczby, napisy itp.
Każda zmienna ma swoją nazwę, która pozwala na jej jednoznaczną identyfikację.
Zmienna posiada również swój typ, który określa rodzaj danych, jakie przechowuje.
W celu utworzenia zmiennej należy ją **zadeklarować**, tzn. podać nazwę oraz początkową wartość.

```
var nazwaZmiennej = wartość;
deklaracja          inicjalizacja
```

Zadeklarować zmienną można tylko raz. Zmienna [przechowuje tylko jedną wartość.
Ale później można inicjalizować ją kilkukrotnie.
Na jednej zmiennej można wykonywać kilka inicjalizacji.
Wartości napisane niżej w kodzie nadpisują te w wyższych linijkach.

Java Script rozróżnia małe i duże litery.

- Zmiennej nie można zaczynać od cyfry.
- Zmiennej nie można zaczynać od wielkiej litery. Zawsze zaczynamy od małej litery.
- Zmiennej nie powinniśmy tworzyć nazw z polskich liter.
- Zmienna dwuczłonowa nie może zawierać spacji. Słowa możemy rozdzielić podłogą (nie można zastosować żadnego innego znaku). Możemy też użyć zapisu wielbłądziego.
- Zmiennej nie można tworzyć ze słów, które coś oznaczają dla Java Script

Wartości zmiennych:

- Ciąg znaków zapisujemy w cudzysłowie
- Liczby jako wartość zapisujemy bez cudzysłowów. W liczbach rzeczywistych separatorem jest kropka.

Typy danych przechowywanych w zmiennych:

Liczba - number,
Ciąg znaków – string
Logika – boolean (true, false)
Obiekty
Specjalny – null, undefined

Sprawdzamy w konsoli typ zmiennej:

```
console.log(typeof(zmienna2));
```

Na zmiennych typu number można wykonywać działania matematyczne.
W zmiennych typu string znak plusa + skleja słowa w jedno słowo

Dynamiczne typowanie zmiennych.

JavaScript sam sobie odgaduje jakiego typu dane są w zmiennych.

Modyfikowanie wartości zmiennych

Zawsze zmiennej można przypisać poniżej zupełnie nową wartość.

```
var zmienna = 1;
zmienna = 2;
```

Można też wartości zmiennej przypisać inną zmienną

```
var zmienna1 = a;
zmienna2 = zmienna1;


var a = 1
var b = a
```

Można nadpisać wartość innej zmiennej

```
var zmienna1 = a;
vzmienna2 = b;
```

Wykonywanie operacji

- Operatory arytmetyczne – mnożenie, dzielenie, dodawanie, odejmowanie.
  Modulo = reszta z dzielenia 14%4 modulo=2 (ile 4 mieści się w 14? I ile brakuje do 14?)

- Operatory inkrementacji i dekrementacji

Ogólnie:
var liczba = 10;
liczba = liczba + 5 to to samo co
Liczba += 5
(wynik 15)

A Inkrementacja (zwiększanie):

```
var liczba = 10;
liczba++; – Najpierw wyświetlamy liczbę, a potem zwiększamy o 1
(wynik 11)
```

```
var liczba = 10;
++liczba; Najpierw zwiększamy liczbę, a potem wyświetlamy wartość
(wynik 11)
```

- Operatory porównania
Służą do porównania argumentów.
== podwójny operator porównania nie bierze pod uwagę typów danych porównywanych
=== potrójny operator porównania uwzględnia w porównaniu typy danych
!= różne od, bez sprawdzania typów
!== różne od, ze sprawdzaniem typów.
<=
>=

- Operatory logiczne
&& - i
|| - lub

Logika.

Prawda = 1
Fałsz = 0

Skojarz && ze znakiem mnożenia
A znak || z dodawaniem

1 && 1 = prawda
1 && 0 = fałsz
0 && 1 = fałsz
0 && 0 = fałsz

1 || 1 = prawda
1 || 0 = prawda
0 || 1 = prawda
0 || 0 = fałsz

- Operatory przypisania
Pojedynczy znak = jest operatorem przypisania

Zapis gravisowy.

var dzień = 20;

var miesiąc = „wrzesień”;

var rok = 2022;

Zamiast pisać:

```
var data = „Dzisiaj mamy ”+ dzień + „ „ + miesiąc + „ „ + rok”;
```

Można zapisać zapisem gravisowym:

```
var data = `dzisiaj mamy ${dzień} ${miesiąc} ${rok}`;
```

Instrukcje warunkowe if

Warunek – rozwidlenie kodu.
Prawda – jedna funkcja, a jeśli fałsz – inne funkcje.

```
if(warunek){
      wykonaj coś;
      i coś;
      i coś;
}
else if(warunekinny){
```

```
      Wykonaj coś;
}
else{
      wykonaj coś innego;
}
```

Jeśli znaleziona zostanie prawda, to nie są wykonywane inne, poniższe instrukcje, elsy itp.

Ify mogą być zagnieżdżane w innych ifach. Czyli całe instrukcje warunkowe mogą być zagnieżdżane w innych instrukcjach warunkowych. 3 stopień zagnieżdżenia to już za dużo wg dobrych praktyk – tworzy dużą nieczytelność kodu. Wtedy trzeba zastanowić się czy nie przebudować kodu.

parseInt, parseFloat

Var liczba1 =

prompt(„textjakis”) – prompt zawsze przyjmuje wartości jako typ string.

Żeby to zmienić na liczbę całkowirtą, musimy wpisać:
parseInt(prompt(„podaj liczbe 1:”));

Żeby to zmienić na liczbę rzeczywistą, musimy wpisać:
parseFloat(prompt(„podaj liczbe 2:”));

Instrukcje wyboru switch

Wyświetl, który kwartał roku prezentuje dany miesiąc:

```
var miesiąc = „luty”;

Switch(miesiąc) wartość poszukiwana {
Case „styczeń”:
      Console.log(„1 kwartał”);
break;
Case „luty”:
      Console.log(„1 kwartał”);
break;
```

```
Case „marzec”:
     Console.log(„1 kwartał”);
break;
default:
     Console.log(„błedny miesiąc”);
}
```

Switch umożliwia grupowanie case'ów.
Po grupie case'ów dopiero występuje console.log i break.
Tylko jeden default może być w instrukcji wyboru. Jak else w ifie.

Operator warunkowy trójargumentowy

Pozwala na ustalenie wartości wyrażenia, w zależności od prawdziwości danego warunku.
Może zastępować ifa w krótkich sprawach

Warunek?wartosc1:wartosc2;

Wzorzec: (warunek) ? prawda : fałsz;

```
var liczba = 5
var wynik;
     wynik = (liczba %2 ==0)? „parzysta” : „nieparzysta”;
console.log(wynik);
```

Pętle

Zostały stworzone po to by móc przetwarzać więcej danych.
Zrozumienie wymaga jeszcze bardziej myślenia abstrakcyjnego, wyobraźni.

Pętle pozwalają na wielokrotne wykonanie tych samych instrukcji.

Rodzaje pętli w js:

- for
- while
- do…while

Pętla for

Pętla for musi wiedzieć ile razy ma się kręcić.

```
for(1rzecz; 2 rzecz; 3 rzecz)
```

Pętla działa na zakresach liczbowych

1 rzecz = od ilu mam się zacząć kręcić `var i=0`

2 rzecz = warunek `i<=10` dopóki i będzie większ od zera, pętla ma się kręcić (sprawdzanie true & false)

3 rzecz = o ile ma się zwiększać / zmniejszać `i++`

0 jest też obrotem. Pierwszym
I jest parametrem pętli. Jest zmienną wyłącznie pętli i jej niezbędnym składnikiem. Bez niej nie działa.
Nie możemy użyć i jako zmiennej w programie. Poza pętlą. Tylko między klamrami możemy wykorzystać i.
Nie rób przypisania na i (i=…)

```
for(var i=1; i<=10; i++){
      console.log (….);
}
```

startuję od 1; czy 1 jest mniejsze od 10 ; (tak)
więc wykonuję instrukcję console.log.

Potem kręci się pierwszy raz (i++)
I sprawdza czy może kręcić się następny raz (sprawdza znów warunek i<=10)
Jeśli tak – kręci się drugi raz

Nie można wpływać na i i wykonywać operacji na i. Nie wolno modyfikować i.
I jedynie można podglądać.

Pętla while

Pętla while jest wykorzystywana wtedy gdy nie wiemy ila razy pętla ma się kręcić.

```
var i=1
while(warunek){
console.log(i);
i++
}
```

dopóki (nie zostanie spełniony warunek)

rób coś

i wyświetl wynik

Tu nie mamy pewności, że zostanie wykonana jakakolwiek instrukcja

Zapis losowania liczb w js: domyślnie startuje od zera, żeby to zmienić od 1 wpisujemy +1

```
Var los = Math.floor((Math.random() *10) +1);
```

```
do{
Coś
}
while(warunek)
```

zrób coś raz

dopóki nie zostanie spełniony warunkunek

Tu instrukcja zostanie wykonana przynajmniej raz

Do pętli jest możliwość użycia break. Możemy przerwać wykonywanie pętli w momencie spełnienia warunku break.

```
For(var i-1; i<100000000000000; i++){
      if(i==5){
      break;
      }
}
```

Pętla zostanie przerwana po 5 obrocie.

Można więc ugryźć kod z innej strony:

```
while(true){ // idealna pętla nieskończona
      var los = Math.floor((Math.random() *10) +1);
      console.log(los);
      if(los == 6){
      break;
      }
}
```

Czyli wykonuj nieskończoną pętlę do momentu dopóki wylosowana zostanie szóstka. Wtedy przerwij pętlę.

Oprócz break, do sterowania pętlą służy nam też continue.
continue blokuje wykonywanie wszystkich instrukcji, ale mówi pętli by obracała się dalej.
Można pominąć więc jeden krok pętli.

Funkcje

Funkcja to wydzielony kawałek kodu, który coś robi. Uruchomi się on tylko wtedy kiedy będziemy chcieli, żeby się uruchomił. Czyli nie automatycznie.
O funkcjach często się mówi, że są to podprogramy. Optymalizuje to pracę programisty.

```
function nazwa_funkcji(){
      coś ma się wykonać
}
```

Żeby funkcja się uruchomiła, trzeba ją wywołać poniżej w kodzie.
Wywołujemy poprzez podanie jej nazwy wraz z nawiasami. Nawiasy rozróżniają funkcję od zmiennych.

```
Nazwa_funkcji(){
      Coś się wykonuje
}
```

W nawiasach funkcji wpisujemy parametry funkcji.
Dzięki parametrowi możemy wprowadzać dane do funkcji. Parametr jest jak zmienna, ale należąca wyłącznie do tej funkcji.
To co się dzieje w funkcji należy wyłącznie do funkcji, nie do programu.
A to co się dzieje w programie nie należy do funkcji i nie może być wykorzystane przez funkcję.

Funkcja może mieć kilka parametrów. Wtedy aby ją wywołać musimy podać dwie wartości przypisane parametrom. Po przecinku.

```
function suma (a,b){
      var wynik = a + b
      console.log(wynik)
}

suma (6,3)
suma (2,4)
```

Jeżeli w wywołaniu funkcji wpiszemy mniej wartości niż parametrów, to musimy utworzyć wartość domyślną dla parametru. Wartość domyśna to 0.

```
function licz (a, b, c, d, e, f=0)
```

Wtedy w miejsce parametru f wartość zostanie wpisana 0.
Jeżeli zabraknie wartości dla jednego z parametrów, to funkcja nie ruszy.
Wartości przypisywane są na parametrach po kolei, od lewej do prawej.

Funkcje w programowaniu dzielą się na dwa typy

- funkcje zwracające
- funkcje niezwracające

Funkcje zwracające

Zwraca daną z wykonanej funkcji do głównego programu.
return.

```
function agnieszka (liczba1, liczba2){
var suma = liczba1 + liczba2;
return (suma);
}
Var ile = agnieszka(3, 8);
If (ile % 2 == 0){
Console.log(„parzysta”);
}
Else {
Console.log(„nieparzysta”);
}
```

return przerywa funkcję. Zwraca wynik i koniec. Działa trochę jak break w pętli.
Dobrą praktyką jest stosowanie jednego returna w funkcji.

Funkcje niezwracające

Nie zwraca danej z wykonanej funkcji do głównego programu

```
function damian (liczba1, liczba2){
var suma = liczba1 + liczba2;
console.log(suma);}
```

Notacja strzałkowa funkcji JavaScript (ES6)

```
nazwaFunkcji = () => {alert(„Hello”)}
```

Jeśli funkcja ma tylko jeden parametr, to można pominąć nawiasy dla parametru.

```
nazwaFunkcji = a => {return a}
```

Jeżeli funkcja ma tylko jedną instrukcję i jest to instrukcja return, to można pominąć nawiasy klamrowe dla instrukcji

```
nazwaFunkcji = a => return a
```

Zmienne ECMAScript JavaScript (ES6)

W ES6 zamiast var zostało wprowadzone `let` i `const`

Hoisting to podnoszenie deklaracji zmiennych

Let – wyłącza hoisting – pilnuje, by kod był wykonywany nadal od góry do dołu.
Czyli nadal najpierw trzeba zadeklarować, utworzyć zmienną, by poniżej jej użyć.
Zmienna jest tylko widoczna w zakresie, bloku, w którym została utworzona.
Niewidoczna jest poza blokiem (np. poza klamerkami, w których została zadeklarowana).

Const – określenie zmiennej, która będzie przechowywała stałą wartość.
Dobra praktyka mówi, że nazwy zmiennej o stałej wartości piszemy wielkimi literami.
Stałej wartości nie można zmieniać, nie wolno przypisywać jej innej wartości.
Deklaracja i inicjalizacja musi nastąpiś w jednym wierszu.
Const nie działa na zmiennych z wieloma wartościami (tablice). Zadziała tylko z jedną wartością prostej zmiennej.

Typy sekwencyjne napisowe JavaScript

Zmienna typu string zachowuje się trochę jak tablica. Ale nią nie jest
W wyrazie, każdy znak ma przypisany swój indeks i możemy zapytać o każdy znak z osobna używając indeksu. Nie można jednak zastąpić żadnego znaku innym. Dlatego jest to niemutowalna tablica.

Wartość lenght zlicza ilość znaków w stringu.

Tablice

Przechowywanie wielu informacji na jednej zmiennej

Tablice traktujemy jak komodę. Do każdej szufladki można coś wrzucić.
Każda szufladka ma nadany numerek. Numery generowane są automatycznie. Zaczynają się od zera.
I nigdzie tego nie widać.

```
let imiona = [„agnieszka”, „maciej” , „kacper”];
```

agnieszka to szufladka 0, maciej to szufladka 1, kacper to szufladka 3

```
console.log(imiona[0])
```

W tablicach występuje inna organizacja pamięci.

Referencja.
Przy tworzeniu tablicy w pierwszym wierszu tworzone są dwie komórki pamięci.
Jedna reprezentuje zbiór danych, a druga przechowuje wartości (referencja to połączenie między nimi).
Ten zbiór wartości nie jest stały, może się zmniejszać, zwiększać. Dlatego jest w osobnej komórce pamięci.

Tablice to struktury danych pozwalające na przechowanie uporządkowanego zbioru elementów.

Tworzenie tablic można wykonać na 2 sposoby

1. Utworzenie tablicy pustej

```
let tab1 = []
lub
let tab2 = new Array();
```

2. Utworzenie tablicy wstępnie zapełnionej informacjami

```
let tab3 = [11,22,33,44];
```

Tablice w JavaScript mogą być zapełniane różnymi typami danych.
Najczęściej będziemy się spotykać z tworzeniem pustej tablicy.

Dodawanie elementów do tablicy

Dodawanie na koniec tablicy:

```
Var imie = [„Ola”, „Tomasz”, „Wojtek”]
Imie.push(„Jola”);
```

Dodawanie na początek tablicy:

```
Imie.unshift(„Jola”);
```

Usuwanie ostatniego elementu tablicy:

```
Imie.pop();
```

Usuwanie pierwszego elementu tablicy:

```
Imie.shift();
```

Usuwanie dowolego elementu tablicy:

```
Imie.splice(1,1);
```

Pierwsza wartość w nawiasie to indeks, a druga to ile elementów wycinamy

Dodawanie dowolego elementu tablicy:

```
Imie.splice(1,0,”Robert”);
```

Pierwsza wartość w nawiasie to indeks, a druga to ile elementów wycinamy, trzecia – co wstawiamy w ineks

Jak odczytywać tablice

Wystarczy podać indeks żądanego elementu w nawiasie kwadratowym.

W odczytywaniu danych z tablicy pomocna może być pętla.

```
tab.length
```

To jest długość tablicy. Można wprowadzić tę wartość do pętli for – ile razy ma się obracać.

Pętla dla tablicy – prostsza

```
let tab = [1,2,3,4]
for (let x of tab){
      console.log(x) (value)
}
```

Na x-a wpadają kolejne wartości. Ale bez indeksów.

A dla czytania konkretnego indeksu inna odmiana tej pętli

```
let tab = [1,2,3,4]
for (let x in tab){
      console.log(x, tab[i]) (indeks, tab[index])
}
```

Pętla w pętli

Pętlę można zagnieżdżać w innej pętli.
Jednak pętla umieszczona w środku innej musi mieć inne parametry sterujące. Inaczej wejdą ze sobą w konflikt.

```
for (let i=0; i<5; i++){
      for (let j=0; j<5; j++){
      console.log(i,j);
}
```

Pętla wewnątrz uruchamia się po pierwszym uruchomieniu zewnętrznej pętli.
I sobie działa swoją ilość razy. Jak się skończy – zewnętrzna pętla uruchamia się 2-gi raz

Zbiory JavaSript ES6

Jest to typ sekwencyjny nieuporządkowany. Rzadziej stosowany niż tablice.
Nie można tu się odwołać poprzez indeks, żeby dostać się do jakiejś wartości. Tu przewagę mają tablice.

Jednak zbiory mają tę przewagę nad tablicą, że gwarantują niepowtarzalność elementów w zbiorze.
Wartości w zbiorze się nie powtarzają.

Tworzenie zbioru

```
let zbior = new Set();
```

Dodawanie do zbioru

```
zbior.add(3);
zbior.add(13);
zbior.add(„d”);
zbior.add(x);
zbior.add(true);
```

Edycja wartości nie istnieje. Bo nie ma jak się odwołać po indeksach.
Możemy jedynie usunąć element ze zbioru i następnie dodać nowy.

Usuwanie ze zbioru

```
zbior.delete(13);
console.log(zbior);
```

Przeglądanie zbioru.
Pętlą let of. Gdyż pętla in wymaga podasnia indeksu, a tutaj nie mamy indeksów

```
for(let x of zbior){;
console.log(x);
```

Rozmiar zbioru

```
console.log(zbior.size);
```

Sprawdzanie czy dana wartość istnieje

```
console.log(zbior.has(„a”));
```

Mapy JavaSript ES6

Typ sekwencyjny uporządkowany. Tak jak tablica.
Mapa pozwala na użycie kluczy w miejsce indeksów stosowanych w tablicy.
Np. zamiast numeru indeksu można wpisać słowo (klucz).

Są klucze (odpowiedniki indeksów) i wartości

Tworzenie mapy

```
let sklep = new Map();
```

Dodawanie do mapy

```
sklep.set(„sok”, 2.22);
sklep.set(„woda”, 3.33);
sklep.set(„cola”, 4.44);
```

Klucze w mapie nie mogą się powtarzać. Inaczej wartości zostaną nadpisane.
A więc podanie tego samego klucza edytuje jego wartość i nadpisuje nową.

Pobieranie wartości z mapy

```
Console.log(sklep.get(„sok”));
```

Odczytanie całej mapy odbywa się za pomocą pętli of

```
for(let x of sklep.keys()){ // tworzy się tablica z kluczami mapy
     console.log(x, sklep.get(x);
     )
```

Sprawdzenie rozmiaru mapy

```
console.log(sklep.size);
```

has to Sprawdzenie klucza mapy, nie wartości

```
console.log(sklep.has(„garnek”));
```

Usuwanie z mapy

```
sklep.delete(„sok”);
```

3. JAVA SCRIPT I HTML

Gdzie w html podłączamy plik js?
Podłączamy go w sekcji head dokumentu. Ale trzeba wstrzymać ładowanie kodu, dopóki nie załadowana zostanie sekcja body. Bo wszystkie elementy strony znajdują się w sekcji body. Więc najpierw muszą pojawić się elementy strony, by na nich mogły zostać wykonane jakieś skrypty.

Trzeba dodać do js. eventListener, który sprawdzi czy załadowana już została cała struktura dokumentu html.

```
dokument.addeventListener(„DOMContentLoaded”, function(){
      dokument.querySelector(„button”).onclick = function(){
      alert(„yeaaa!”);
      }
}
```

Odwołanie się do selektora <button> w html

DOM – struktura dokumentu html (zagnieżdżeń itp.) – Dokument Object Model

Formularz

Pobieranie danych z pola input

Selektor w JavaScript jest zawsze stringiem

Html:

```
<p>
      Imie<br>
      <input type=„text” name=”imie”/>
</p>
```

Js:

```
let imie = document.querySelector(„input[imie]”).value;
alert(imie);
```

Pobiera daną zmienną z pola tekstowego i wyświetla wpisaną wartość w polu alert.

Inputy i checkboxy – czyli pola zgrupowane – to będzie wyglądało już inaczej.

html

```
<p>
      Wybierz kolor<br>
      <input type=„radio” name=”kolor” value=”bialy”>Biały
      <input type=„radio” name=”kolor” value=”czerwony”>Czerwony
</p>
```

Js:

`querySelectorAll` – tworzy tablicę z wszystkimi selektorami o danym typie.

```
let kolory = document.querySelectorAll(„input[type=radio]”);
let kolor;
for(let x of kolory){
      if(x.checked == true){
      kolor = x.value;
      break;
      }
}
```

Jak wstrzyknąć pobrane dane do fragmentu html? innerHTML

```
let imie = document.querySelector(„input[name=imie]”).value;
document.querySelector(#imie).innerHTML = imie;
```

4. PROGRAMOWANIE OBIEKTOWE

Wcześniejszy model programowania nazywamy programowaniem strukturalnym.

Teraz wchodzimy w temat programowania obiektowego.
Programowanie obiektowe zostało wymyślone po to, by odzwierciedzlić, przełożyć na kod świat, który występuje wokół. Miało to ułatwić programiście programowanie.

Obiekt.
Mamy domek. W realu. Zanim domek powstał, musiał powstać jego projekt.
Tak samo jest w programowaniu. Choć akurat w JS możesz wybudować domek bez projektu. Tylko w tym języku. Jest on poprawiany pod tym kątem.
Obiekt powstaje więc na sam koniec łańcucha pewnych zdarzeń (projekt).
W projekcie nie będą interesowały nas szczegóły. Projekt będzie potrzebny tylko do tego, żeby stworzyć obiekt. Z jednego projektu może powstać nieskończenie wiele obiektów.

Literał obiektowy – to budowanie w JS obiektu BEZ PROJEKTU!

```
samolot = {
      nazwa: „Boeing”,
      cena: 1111,
      pulap: 44444,

      lec: function(){
      console.log(„LECE!”);

      laduj: function(){
      console.log(„LADUJE!”);
}
console.log(samolot);
```

W tym przypadku nasz samolot to taka napompowana zmienna.
Tu obiekt powstał BEZ PROJEKTU.

Widzimy wewnątrz coś jak funkcje. Nie jest to jednak do końca funkcja (zwracająca, niezwracająca). Wewnątrz obiektu nazywamy to **METODĄ**.

To samo – nazwa, cena itp. – te wszystkie cechy – są jak zmienna. Jednak wewnątrz obiektu nie jest zmienną w programie, tylko indywidualną cechą obiektu. Taką „zmienną” wewnątrz obiektu nazywamy **POLEM.**

Jeżeli metoda miałaby się jakoś odnieść do pola obiektu, to musimy wpisać przed „zmienną” .this.

Programowanie przy pomocy konstruktora - – to nakładka do ES6, która pozwala budować obiekt za pomocą wcześniejszego projektu PROJEKTU!

Ten projekt określamy w kodzie jako:

```
class Samolot{…
}
```

Nazwę klasy piszemy zawsze dużą literą.

Klasa (nasz projekt) służy tylko do tego, żeby stworzyć z tego teraz obiekt.
Jeżeli nie tworzymy obiektu, to klasa jest nam niepotrzebna.

```
class Samolot {
nazwa;
pulap;
cena;
}
```

Teraz na podstawie naszego projektu utworzymy obiekt:

```
let ob1 = new Samolot();
let ob2 = new Lodz();
```

Klasa podana tu z nawiasami to nie jest funkcja. Dlatego, że funkcje piszemy z małej litery.
Właśnie po to jest to rozróżnienie – klasy są pisanme zawsze wielką literą.

Rozbudujmy obiekt według projektu:

```
let ob1 = new Samolot();
ob1.nazwa = „F15”
ob1.pulap = 9999;
ob.cena = 12.99;

console.log(ob1);
```

Zbudujmy drugi obiekt według projektu:

```
let ob2 = new Samolot();
ob1.nazwa = „Awionetka”
ob1.pulap = 9666;
ob.cena = 11.99;

console.log(ob2);
```

Metoda **KONSTRUKTOR –** ma za zadanie przyspieszyć tworzenie obiektu. Żeby nie było aż tyle pisania przy budowaniu każdego obiektu.
Konstruktor zamieszczamy w klasie – tworzy on automatycznie cechy obiektu.

Na przykładzie naszego projektu samolotu:

```
class Samolot {

      constructor(_nazwa, _pulap, _cena){
            this.nazwa = _nazwa;
            this.pulap = _pulap;
            this.cena = _cena;
      }
}
```

I nasze obiekty mogą wyglądać teraz tak:

```
let ob1 = new Samolot(„F15”, 9999, 12.99);
let ob2 = new Samolot(„Awionetka”, 9666, 11.99);

console.log(ob1);
```

Dziedziczenie

Dziedziczenie dotyczy klas. To klasy dziedziczą po sobie. Nie obiekt. Obiekt tylko korzysta z tego dziedziczenia.

Klasa Samolot i klasa Łódź mają wspólne pola i metody.
Można je ująć w nadrzędną klasę Pojazdy, w której znajdą się wspólne pola i metody.
Klasa nadrzędna bywa nazywana klasą bazową, nadklasą lub klasą parent.
Natomiast klasy podrzędne bywają nazywane podklasami, klasami child, klasami pochodnymi, klasami rozszerzającymi (nadklasę).

Kierunek dziedziczenia.
Potem Samolot dziedziczy po Pojazdach. Łódź tak samo.

```
class Samolot extends Pojazdy{
class Łodz extends Pojazdy{
```

W JS jedna klasa może dziedziczyć tylko po jednej klasie. Nie może dziedziczyć po dwóch klasach.

Może natomiast dziedziczyć kaskadowo.

Przesłonięcie – to inaczej nadpisanie.
Jeśli obie klasy mają identyczną metodę i klasa B dziedziczy po klasie A, to przy tworzeniu obiektu z B tylko metoda z B będzie brana pod uwagę. Ona nadpisze tę odziedziczoną po A. Bo jest bliżej.

Dodanie super.metoda – odwoła się do metody z nadklasy.

```
class A{
      hello(){
            console.log(„AAA”);
      }
}
class B extends A{
      super.hello(){
            console.log(„BBB”);
      }
}
Let ob. = new B();
```

Teraz zostaną wyświetlone obie metody.

Konstruktory w nadklasach i podklasach.
Jeżeli w obu są konstruktory, to zadziała ten, który jest bliżej tworzonego obiektu.
Dla obiektu B zostanie użyty konstruktor z klasy B.

Jak więc użyć konstruktora z nadklasy i z podklasy jednocześnie?

```
class Pojazdy {
      constructor (marka, model, cena){
      this.marka = marka;
      this.kolor = kolor;
      this.cena = cena;
      }
}
class Samolot extends Pojazdy {
      constructor (marka, model, cena, pulap){
      super(marka, kolor, cena);
      this.pulap = pulap;
      }
}
let ob. = new Samolot(„F15”, „szary”, 1234, 9999);
console.log(ob.);
```

Trzeba wyjść od obiektu. Patrzeć czego potrzebuje obiekt.

Czasem klasy tworzy się tylko po to, żeby dziedziczyć, a nie tworzyć obiekty. Po to , żeby oddzielić fragmenty kodu.

MVC – Model, View, Controller

Model – model danych
View – widiok, program w sposób interaktywny pobiera od użytkowanika dane
Controller – przetwarza dane dostarczone przez widok.

Pola publiczne i prywatne

Pole prywatne oznaczamy znakiem #.

Dostęp do pola prywatnego jest możliwy tylko z wnętrza klasy, w której to pole się znajduje.

```
class Bankomat {
      #kasa; //deklaracja pola prywatnego
      constructor(nazwa, miasto){
            this.nazwa = nazwa;
            this.miasto = miasto;
            this.#kasa = 100000000;
}
```

STORAGE

Do przechowywania danych służą albo cookies (gorsze), albo magazyn przeglądarkowy (storage) – lepsze.

Cookies ingeruje w dysk twardy tworząc plik.
Storage tego nie robi.

Storage wygląda jak mapa. Ma klucze i wartości.
Local storage i session storage.

Local przechowuje dane na czas nieokreślony.
Sesja to czas w jakim nasza przeglądarka jest otwarta. Gdy zamknie się przeglądarkę, wszystkie dane są czyszczone.

Jeżeli chcemy dodać coś do storage, to:

```
localStorage.setItem(„newStorage”);
console.log(localStorage.getItem(„newStorage”));
```

Wartości Strage są uniwersalnie stringiem.

JSON – JavaScript Object Notation

Jest to uniwersalny format danych.
Różne dane zapisuje w formie stringów.
Tablice z obiektami, mapy – są przekształcane do formy tekstu i na odwrót: teks przekształcany jest z powrotem do map i tablic.

```
JSON.stringify // przekształca dane na tekst string
JSON.parse // przekształca tekst string na dane
```

Cookies

Ciasteczko nie może powstać bez nazwy, wartości i daty wygaśnięcia.
Posiada swoją ścieżkę, która określa na której podstronie jest zamieszczone. Najczęściej jest to główna strona.

Ważność ciastka podajemy w milisekundach. Musimy najpierw pobrać dzisiejszą datę.
Tworzymy najpierw obiekt z klasy Date.

```
let d = new Date();
d.setTime(d.getTime()+(2*24*60*60*1000));
```

Teraz tworzymy ciastko:

```
document.cookie = "ALX=Kurs JS;expires="+d.toUTCString();
```

Często dane przekazywane są w formie stringa, a poszczególne dane segmenty rozdzielone są separatorami. Żeby wyciągnąć jakąś daną ze stringa:

```
txt = "Adam;Nowak;4327532467;adam@amda.pl";
let x = txt.split(";");
console.log(x[1]);
```

JQuery

Podpinanie biblioteki:
Można bibliotekę pobrać ze strony JQuery lub podlinkować z serwera.
Najpierw należy podlinkować bibliotekę, a poniżej dopiero własny plik js.

```
lkkkkkk.
```

Skrypty JQuery zaczynamy od znaku dolara $.

Teraz zamiast document.adEventListener („DOM..)…
Możemy zapisać krócej:

```
$(document).ready(function(){
)}
```

Selektory działają tu dokładnie jak w CSS tylko muszą być stringiem. Zamiast dwukropków css wpisujemy wartość po przecinku w cudzysłowach.
this piszemy bez cudzysłowów.

Dodanie klasy do znacznika:

```
$(„znacznik”).addClass(„nazwaklasy”);
$(„znacznik”).removeClass(„nazwaklasy”);
```

Pobieranie wartości atrybutu id:

```
$(„p”).click(function(){
let color = $(this).attr(„id”);
})
```

Pętla for each

Służy do odczytywania typów sekwencyjnych

Odczytanie wartości tablicy:

```
Let tab = [22, 33, 44, 55, 66, 77];
tab.forEach(wartość => {
console.log(wartość);
});
```

Odczytanie indeksu i wartości tablicy:

```
Let tab = [22, 33, 44, 55, 66, 77];
tab.forEach((wartość, index) => {
console.log(index, wartość);
})
```

JQuery.ajax

PHP. Technologia backendowa.
JavaScript zaczyna wychodzić poza przeglądarkę i potrafi komunikować się z plikami backendu – na serwerze.

Wychodzimy ajaxem na zewnątrz na serwer do technologii PHP.

node

wyciąga silnik interpretujący JS z przeglądarki i wstawia swój, umożliwiający uruchamianie skryptów poza przeglądarką. Dzięki temu JS staje się też językiem backendowym.

Musi być zainstalowany program nodeowy.

Node ma bardzo dużo bibliotek.
Można postawić nodem serwer. React wymaga nodea.

W wierszu polecenia:
`node -v i enter`

W Vs Terminal – `command prompt`

`npm init` – inicjalizowanie biblotek. Enter wszędzie.
Pojawia nam się plik package.json

instaluj bibliotekę do obsługi i tworzenia serwera: `npm install exspress`

instaluj system szablonów: `npm install ejs`

instaluj obsługę formularzy, wym. danych: `npm install body-parser`

react

instalacja.

Framework w odróżnieniu od biblioteki narzuca sposób pracy. Każe tworzyć konkretną strukturę plików. Żeby w reaccie móc cokolwiek robić potrzebne są: komponenty, propertisy i state'y.

Komponent – to moduł zawierający pewną wydzieloną funkcjonalność.
Komponent w reaccie to klasa, która znajduje się w pliku dokładnie o takiej samej nazwie.

Propsy są read only. Nie można ich zmieniać wewnątrz komponentu. Propsy to to co idzie do komponentu w chwili wywołania.

Zmienne wewnętrzne komponentu nazywane są stateami. Nie są nigdzie przekazywane, w przeciwieństwie do propsów.

Metoda setState zmienia naszego statea i uruchamia rendera

JAVA SCRIPT ZAAWANSOWANY

1. SHELL

W Git bash komendy:

```
$pwd
```

Pokazuje w którym miejscu na dysku jesteśmy. Pokazuje aktualną ścieżkę. Informuje, w którym jesteśmy folderze.

```
$ls
```

Pokazuje jakie mamy dostępne pliki w folderze

```
$ls -l
```

Lista plików w formacie listy

```
$cd desktop
```

Wejście do folderu desktop. Klawisz tab wyświetla podpowiedzi

```
Desktop cd ..
```

Wejście do folderu wyżej (powrót).

```
Clear
```

Wyczyszczenie terminala.

```
mkdir
```

Utworzenie folderu. Po spacji wpisujemy nazwę dla naszego nowego folderu nie zawierającą spacji

```
touch kurs.txt
```

Touch + nazwa pliku z rozszerzeniem tworzy nowy plik

```
vim kurs.txt
```

vim + nazwa pliku otwiera ten plik, żeby go np. edytować. Jest to tryb edycji.
Z trybu edycji wychodzimy klawiszem escape.

```
:wq
```

Zapisuje plik i wychodzi z niego.

2. GIT HUB

Jest to taki dropbox dla kodu.
Przechowujemy kod w repozytoriach.
Git lab to alternatywa dla Git Huba. A także Bitbucket.

Podstawy języka GIT

Pracujemy w Git Bashu. Mamy już założone konto na git hubie.

Upewniamy się, że język GIT jest u nas zainstalowany.

```
git --version
```

Pokazuje jaką mamy zainstalowaną wersję języka GIT.

Żeby skonfigurować GIT-a na komputerze aby poprawnie komunikował się z repozytorium na Git Hubie, musimy najpierw powiedzieć kim jesteśmy.

```
git config --global user.name 'Agapelka'
git config --global user.email 'pelkagnieszka@wp.pl'
```

Zawsze zasada jest taka, że najpierw tworzę repozytorium na Git Hubie, a dopiero potem podłączam go u siebie w git bashu.

Komendy dla GIT:

```
git init
```

Tworzy nowy projekt GIT. Teraz mogę dodawać pliki pliki do repozytorium na serwerze. Ale mój folder lokalny też jest skonfigurowany do działania w GIT

```
git status
```

Sprawdza lokalną wersję i porównuje z tym co jest na serwerze. I wyświetla ewentualne braki.

```
git remote add origin (ścieżka do naszego folderu na Git Hub)
git remote remove origin
```

Podłącza nas do zdalnego repozytorium. Remove odłącza.

```
git remote -v
```

Sprawdza do jakiego repozytorium zdalnego jesteśmy podłączeni

Teraz musimy wysłać brakujące pliki na zdalny serwer.

1. Najpierw dodajemy pliki do paczki (COMMIT)
2. Nazywamy paczkę (commit)
3. Wysyłamy paczkę

Ad.1

```
git add .
```

Dodaje do commita wszystkie pliki z folderu.

Ad.2

```
git commit –m „nazwa commita”
```

Stworzy commita o nazwie ‘nazwa commita’

```
git log
:q (wyjście z git loga)
```

Sprawdza jakie commity zostały utworzone

```
git branch
```

Sprawdza na jakiej gałęzi jesteśmy.
Każdy z teamu programistycznego tworzy własne gałęzie. Commity wysyłamy do własnych gałęzi.

Ale później sami nie mergeujemy ich do gałęzi master, tylko wysyłamy merge request.
Służy to temu, żeby programiści nie wrzucali każdy swoich zmian, dublijąć je, tylko decyduje o tym admin.

```
git push origin master
```

Wysyła paczkę

Pobieranie repozytorium nie naszego na dysk:

```
git clone (ścieżka do folderu na Git Hub)
```

Pobieranie najnowszej wersji repozytorium nie naszego na dysk:

```
git pull origin master (ścieżka do folderu na Git Hub)
```

Git conflict. Jak pobrać zmienioną wersję z repozytorium usuwając własne zmiany?

```
git reset --hard
```

3. DOM

console.log(window); - wbudowany obiekt przeglądarki

console.log(document); - obiekt DOM

Żeby mieć jakiekolwiek interakcje z html – musimy najpierw złapać element – selector.
Czyli najpierw musimy pokazać na czym chcemy zrobić zmianę (złapać element), a potem zrobić zmianę.

```
const heading = document.querySelector('h1');
```

Elementy w JS najlepiej łapać za pomocą #id. Klasy zostawić dla css

Zmiana elementu.

```
heading.innerHTML = '<p>Zmiana tekstu w znaczniku</p>';
heading.outerHTML =
heading.innerText = 'Zmiana tekstu w znaczniku';

Manipulacja klasami
heading.classList.add('nowa-klasa');
heading.classList.remove('nowa-klasa');
heading.classList.toogle('nowa-klasa');

Manipulacja atrybutami (src, alt, value, id)
Console.log(heading.id)
heading.id = 'NowyId'
heading.id += 'DrugieId'
console.log(heading.getAttribute('id'))
heading.setAttribute('id', 'anotherId')
```

Eventy - Zdarzenia na stronie, za które można złapać.

```
- click
- change (keyup, keydown)
- submit
- resize
- DOMContentLoaded
- Focus
```

Jak wpinać eventy na stronie?

```
const handleClick = (event) => { // w nawiasie jest event, nawet
domyślnie. Piszemy jeśli chcemy coś z nim robić dalej

console.log('hej');


document.addEventListener('click', handleClick) // wydarzenie i
funkcja, którą trzeba zadeklarować wyżej.
```

```
event.preventDefault();
```

Ta metoda blokuje domyślne zachowanie przeglądarki (wysyłanie submita na przykład)

Local Storage

W localStorage nie działają struktury JS.

Działa w nim format JSON

```
localStorage.setItem('text', 'hello');

console.log(localStorage.getItem('text');
```

```
JSON.stringify(); // zamaina JS do JSON

JSON.parse(); // zamiana JSON do JS
```

Najpierw trzeba odczytać dane, które mamy przesłać do localStorage:
To dosy , lista książek są zawsze tablicą obiektów

```
const todos = [

{
     title: 'Wynieść śmieci'
```

```
  },
  {
    title: "Pozmywać"
  }
];
```

Zamiana na JSON:

```
localStorage.setItem('todos', JSON.stringify(todos))
```

Wzór na przechodzenie przez elementy pętlą forEach

```
todos.forEach(todo => {
  todoList.innerHTML += `
    <li> ${todo.title} </li>
  `
```

4. PROGRAMOWANIE FUNKCYJNE

Służy do operacji na tablicach (w celu pisania aplikacji)

Funkcje:

1. forEach
2. map
3. find
4. filter
5. * reduce

Schemat pisania tych funkcji:

```
[].metoda(function(element tablicy)){
Console.log(elementTablicy)
})
```

Notacja strzałkowa:

```
[].metoda((element tablicy) =>{
Console.log(elementTablicy)
})
```

forEach

Służy do przechodzenia przez elementy i np. ich wyświetlania na ekranie lub w konsoli. Można też liczyć nią sumy czy wykonywać obliczenia

```
Books.forEach((book, index) => {
console.log(`{book.title} o indeksie ${index}`);
})
```

Liczenie sumy:

```
 let sum = 0;
books.forEach((book) => {
sum += book.price
});

Const average =  sum/books.lenght;
Console.log(average);
```

map

Służy do tego, żeby każdy element tablicy zamienić na coś innego. Map = przechodzimy przez wszystkie elementy i zwracamy tylko to co chcemy.

Wewnątrz funkcji map **zawsze musi być return.**

Jest to funkcja zwracająca

```
const newBooksWithOnlyTitles = books.map(book => {
return book.title
});
Console.log(newBooksWithOnlyTitles);
```

find i filter

Metoda find służy do wyszukiwania pierwszego elememtu z tablicy, który spełni dane kryterium.
Metoda filter zwraca nową tablicę, zawierającą elementy, które spełniają dane kryterium.

```
const puzoBook = books.find(book => {
     return book.author.includes('Puzo')
});
```

Includes – zwraca elementy zawierające dany string również tablice.
Jeśli find nic nie znajdzie to zwróci undefined.

Filter – zwraca drugą tablicę – z wszystkimi elementami spełniającymi warunki.

```
const booksBeforeMillenium = books.filter((book) => {
return book.year > 2000
});
Console.log(booksBeforeMillenium);
```

sort

To jest snippet.
Jest to wbudowana funkcja do sortowania bąbelkowego.

```
const sortedBooks = books.sort((a, b) => {
return a.price – bprice ? -1 : 1;
});
```

Funkcja przechodzi przez tablicę i porównuje elementy ze sobą pod kątem kryterium.

```
? -1 : 1;
     // To jest shorthand na if else
if(a.price – bprice) {
return -1;
}else{
Return 1;
```

Jeśli chcemy mieć kolejność od największego do najmniejszego, to musimy zamienić kolejnością 1 i -1.

Jeśli chcemy przesortować po innym kluczu, musimy pamiętać żeby zmienić ten klucz (tutaj price)

Reduce

```
const sum = books.reduce((accumulator, book) => {
console.log(accumulator);
return accumulator + book.price;
```

accumulator to wartość początlkowa. Na początku jest 0 i potem robi książkę.
W następnym działaniu do książki accumulator dodaje kolejne wywołanie – cenę kolejnej książki.
Przy następnym wywołaniu dodaje do zapamiętanej sumy cenę ostatniej książki.
I kończy, bo nie mam więcej książek.

4. BIBLIOTEKI NPM

npmjs.com/package/

Są to paczki js.

Komendy node:

`Node -v` – sprawdzenie wersji node

`npm init` – podłączenie biblioteki npm

`npm install nazwapaczki`

`npm install` – instaluje całe paczki zapisane w pliku package.json – jeśli dostaliśmy projekt od kogoś

`npm run nazwaskryptu` – uruchomienie skryptu znajdującego się w package.json

Biblioteki

package-lock json – zawiera informacje o tym co zostało zainstalowane, jakie paczki

node-modules – tu znajdują się zainstalowane paczki

Trzeba utworzyć plik .gitignore i napisać w nim co ma ignorować git. Czyli napisać node_modules.
Bez tego git pokazuje, że ma ponad 3 tys zmian – to są wszystkie paczki i wszystkie pliki zainstalowanych node modules.

W pliku package.json musimy wpisać
source:index.html

```
  "description": "",
  "main": "main.js",
  "source": "index.html",
```

A w skryptach:

```
  "scripts": {
    "dev": "parcel"
  },
```

Robimy npm run dev

Robimy plik htm z podstawową strukturą i odpalamy w przeglądarce.
Na pasku wpisujemy: localhost:1234
Mamy nasz serwer.

Biblioteka parcel

Dzięki temu, że używamy parcela, to możemy wpisać do znacznika script w htmlu atrybut type do znacznika script. type = „module”

```
    <script src="js/main.js" type="module"></script>
```

Mogę podzielić teraz paczką parcel kod js na mniejsze części w osobnych plikach.

Np. w pliku main.js możemy napisać funkcję, a jej wywołanie napisać w innym pliku.
W tym celu w main przed nazwą zmiennej wpisujemy `export`
a w pliku z wywołaniem wpisujemy:

```
import { nazwafunkcji } from './nazwaplikujs';
```

dobra praktyka to umieszczać importy z bibliotek na górze, a pod nimi dopiero importy z plików.

Ctrl +c – w git bashu – zatrzymuje serwer.

Jeżli chcemy coś zainstalować nowego, to musimy zatrzymać nasz serwer.

5. ASYNCHRONICZNOŚĆ I PROMISE

JS synchroniczny:

Console.log(‘Pierwsza akcja’);

Console.log(‘Druga akcja’);

Console.log(‘Trzecia akcja’);

JS Asynchroniczny

Czyli nie wykonuje się od razu, tylko po jakimś ustalonym czasie

Node.JS – odczyt, zapis pliku

JS funkcje czasowe:

```
setInterval
```

to funkcja, która wykonuje się co jakiś czas (zdefiniowany)

SetTimeout

```
Console.log('Czynność 1');
//setTimeout(nazwaFunkcjiOpóźnianej, opóźnienieWMilisekundach)
czyli
setTimeout(() => { //- ta funkcja wkładana jest na koniec stosu
console.log('Czynność 2');
}, 1000)
Console.log('Czynność 3');
```

to funkcja, która odpali się raz, po jakimś opóźnieniu.
Jest wkładana na koniec stosu.

Promise (http)

to wbudowany obiekt JS, służący do obsługi asynchroniczności.

HTTP – internet -
Każdy komputer podłączony do sieci mas swój adres – adres IP.
IPv4 i IPv6 (16 cyfr).

Adresy DNS – które mówią nam jaka domena jest przypisana do danego IP. Dzięki temu widzimy adres domenowy zamiast cyferek IP.

Komputery w sieci muszą się ze sobą komunikować.
Komunikacja pomiędzy moją przeglądarką a serwerem to komunikacja http.
HTTP to standard w komunikacji. Zasady komunikacji między komputerami. Hypertext Type Text Protocole. Komunikacja odbywa się na zasadzie zapytań i odpowiedzi. **Request i response.**
Request zawsze musi iść pierwsze. Przeglądarka wysyła zapytania do serwera i czeka na jego odpowiedź.

Istnieje też http push – gdzie to serwer inicjuje połączenie.

Request:

Nagłówki HTTP – możemy w nich wykonać metody:

GET – prośba o pobranie danych z serwera

POST – prośba o wysłanie danych do serwera

DELETE – prośba o usunięcie danych z serwera

PUT – prośba o edycję danych z serwera (ew. jeśli nie ma danego zasobu, tworzy go)

PATCH - prośba o edycję danych z serwera (tylko do zmiany istniejącego zasobu)

Response:

HTTP Status Codes:

404 – Not Found (prosimy o dane, ale serwer nie może tego znaleźć)

200 – OK (ok, zrozumieliśmy twój request, oto wynik)

201 – OK, created (twój request jest ok i utworzyliśmy nowy zasób)

500 – Server Error – (twój request jest ok, ale serwer działa nieprawidłowo)

301 – Moved Temporarily -

302 – Moved Permanently

HTTP Headers

To dodatkowe informacje, które są wysyłane na serwer.
Przykładem jest np. Content-Type.

Content-Type – informuje serwer, jaki format danych wysyłamy lub jaki format danych oczekujemy.

Authorization – przekazujemy do serwera token (JWT Token), który autoryzuje nas jako uprawnionych do korzystania z serwera. Np. login i hasło. Wysyła się na backend.

HTTP Parameters:

W parametrze przekazujesz to czego dokładnie chcesz.

GET - params (queryparams)

POST – body (obiekt, pod którym są zawarte wszystkie dostępne rzeczy)

PUT – body

DELETE – params (queryparams)

W folderze z projektem tworzymy folder „server”
W nim plik data.json.
Będziemy w nim przechowywać dane – zamiast w localStorage będę teraz na serwerze.
Dane do przekazania serwerowi wpisujemy do pliku data.JSON.
Czyli w przypadku chatu – kopiujemy tam zawartość localStorage – zapamiętane nasze wiadomości.
Teraz potrzebujemy zainstalować paczkę, za pomocą której uruchomimy ten plik.
Będzie to npm server.

W pliku package json uruchamiamy nasz skrypt serwera:

```
"sever": "json-server ./server/data.json -p 5000"
```

W terminalu odpalamy skrypt komendą – npm run server (w moim folderze)
Server zostanie uruchomiony w porcie 5000 localhost.

Ponbieważ mamy teraz dwa skrypty (i dwie paczki) – musimy zainstalować kolejną paczkę, która pozwoli na odpalanie obu rzeczy na raz:
npm install npm-run-all
W pliku package json uruchamiamy nasz skrypt npm-run-all:

```
"start": "npm-run-all --parallel dev server"
```

W terminalu odpalamy skrypt komendą – npm run start (w moim folderze)

W przeglądarce odpalamy localhosty w dwóch oknach: 5000 oraz 1234 (z chatem)

Jak korzystać teraz z biblioteki?
W pliku chat.js:

```
Fetch('http://localhost:5000/messages')
// ścieżka do naszych danych w localhoście (json.data)
```

Fetch to wbudowana metoda obiektu window służąca do zapytań http.
Domyślną metodą zapytań jest GET.

Odpowiedź dostaniemy jednak z opóźnieniem iluś milisekund.
Mamy więc **PROMISE** – obiekt wbudowany do asynchroniczności.

Stany Pomise:

- pending – promise jest w trakcie wykonywania
- fullfiled – promise został wykonany pozytywnie
- rejested – promise został wykonany i się nie udał

```
Dopisujemy do powyższego kodu:
Fetch('http://localhost:5000/messages')
      .then ((response) => {
return response.json()
})
.then((data) => {
console.log(data);
})
catch((error) => {
console.log(error);
})
```

Metoda response.json – wbudowana metoda w fetch, służy do wyłuskania wartości zapytania http.

Błędy z promisa nie są wyświetlane w konsoli, do tego potrzebujemy powyższej metody catch.

SNIPPET DO ZAPYTAŃ http:

```
fetch('link)
      .then ((response) => {
return response.json()
})
.then((data) => {
data.forEach((message)=> { //wklejamy forEacha który robił LoStorage
renderMessage(message);
})
catch((error) => {
console.log(error);
})
```

Teraz musimy usunąć w kodzie fragmenty wczystywania i odczytywania z LocalStorage.

Bo teraz zamiast LocalStorage korzystamy z Promise z zapytania HTTPS.

Powinno wyświetlać teraz wiadomości, które mamy w json.data.

A teraz wysyłanie danych z serwera do LocalStorage.

```
const messageToSend = {
id: '1',
message: 'treść wysłana do serwer',
Author: 'Damian'
}

fetch('http://localhost:5000/messages')
method:'POST'
headers:{
'Content-type': „application/json"
},
body: JSON.stringify()
})
```

Id nie może się powtarzać!

Inaczej wywali nam error 500.

Potrzebujemy więc tworzyć unikalnych id.
Do tego służy paczka npm UUID Id.
Trzeba ją zainstalować.

Snippeta fetcha wrzucamy do funkcji handle submit, a do obiektu newMessage wpisujemy pole id o wartości uuidv4. Rozbimy porządek z fragmentami dot. LocalStorage.