* **Event** – signal generat de un DOM Node
* **Daca folosim this in functia eventului, this mereu se refera la elementul de la care porneste eventul**

**A 2 metoda de a adauga event(veche)**

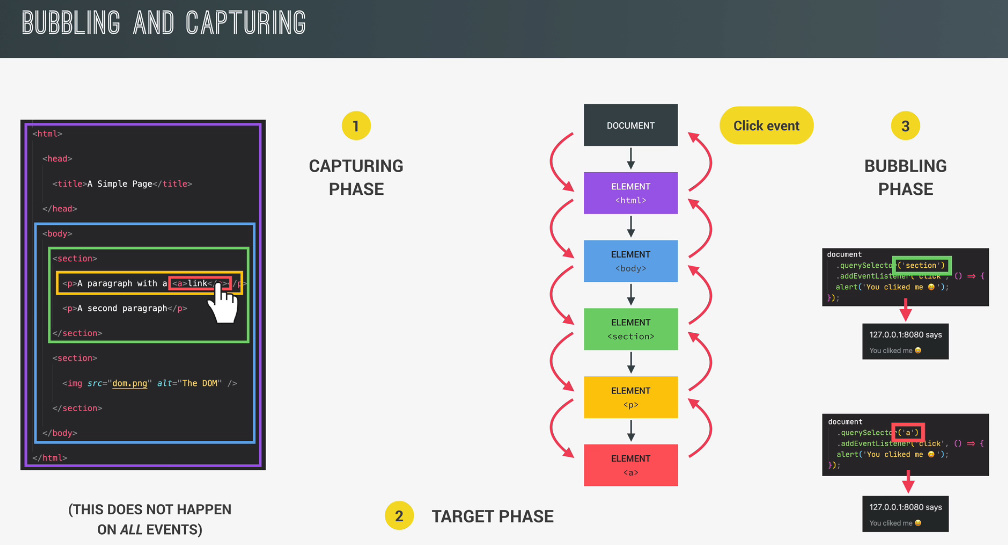
* Metoda data permite de a adauga un singur event per action
* **element.action = function()**
* *btnScrollToInfo*.onclick = () => {  
   *infoSection*.scrollIntoView({behavior: 'smooth'})  
  }

**Stergerea unui evenListener**

* .**removeEvenetListener(‘unde’, function)**
* const func = function() {  
   alert('Hello!!');  
   *infoSection*.removeEventListener('mouseover',func);  
  };  
    
  *infoSection*.addEventListener('mouseenter', func)

Deci, va trebui sa pastram referinta functiei, de altfel nu putem sa o scoatem din eventListener

**Event Propagation: Bubbling and Capturing**



Cand setam un event pentru un element, propagarea lui are loc de la parinti la el si de la el la parinti

1. Cand facem click pe <a>, se creaza un event
2. Eventul intai incepe din Document, apoi merge pe la fiecare parinte pana ajunge la <a>
3. Odata ce a ajuns la <a>, se duce inapoi prin fiecare parinte pana la Document, adica root

* Asta e important, deoarece un event pus unui element e pus automat si la toti parintii sai, desi nu chiar mereu.
* **Capturing** – propagarea de la parinti la element
* **Bubbling** – propagarea de la element la parinti inapoi

**Exemplu**

Fie ca avem un asa code html:

<style>  
 body{  
 text-align: center;  
 }  
 #div1{  
 width: 800px;  
 height: 800px;  
 }  
 #div2{  
 width: 400px;  
 height: 400px;  
 }  
 a{  
 width: 200px;  
 height: 200px;  
 }  
</style>

......  
<div id='div1'>  
 <div id='div2'>  
 <a href='#' id='a'>Click</a>  
 </div>  
</div>

- div2 si a sunt copiii lui div1

- a este copilul lui div2 si div1

- div1 e parintele lui div2 si a

- div2 e copilul lui a

Datorita la event propagation, un event efectuat asupra unui element face ca acest event, de ex, ‘click’, sa fie efectuat si asupra la toate elementele parinti.

Insa, un event, de ex ‘click’, ce are loc asupra unui parinte, nu se propaga si asupra la copii!

* Odata ce de ex, un ‘click’ a avut loc pe element a, ‘click’ va porni de la Document node, si va ajunge pana la a trecand prin toti parintii, insa nu se realizeaza inca eventul. Asta e **capturing**
* Odata ce event a ajuns la elementul a, are loc bubbling, si deci anume acum se executa eventul asupra lui a, si apoi asupra la parintii lui, deci asupra lui div1 si div2, insa evident nu si boddy
* Ideea e simpla, a se afla si in div1 si div2, caci si div2 se afla in div1, si deci apasarea lui a ar fi ca si cum s-ar apasa toate 3, asta e tot
* Deci, la toate 3 elemente se va inregistra ‘click’ event, insa daca ele au un handler pentru ‘click’ event, asta deja e treaba lor
* function addColor(e){  
   this.style.backgroundColor = `rgb(${*Math*.floor(*Math*.random() \* 255)},  
   ${*Math*.floor(*Math*.random()\* 255)},  
   ${*Math*.floor(*Math*.random() \* 255)})`;  
    
   *console*.log(this)  
    
  }  
    
  *document*.getElementById('div1').addEventListener('click', addColor);  
  *document*.getElementById('div2').addEventListener('click', addColor);  
  *document*.getElementById('a').addEventListener('click', addColor);

mare atentie sa folosim this cand vrem ca functia sa se refere la elementul actual ce o apeleaza!

Daca am folosi **e.target.style.backgroundColor**, e.target mereu va fi elementul care a pornit eventul, adica a, indiferent care ar fi elementul actual ce a apelat functia. Am folosi asta daca nu am vrea ca efectul sa fie aplicat asupra la toate 3 elemente, ci doar asupra la cel ce a apelat eventul,adica a

* O alternativa pentru this, e si **e.currentTarget,** care e elementul curent, nu cel ce a apelat eventul
* function addColor(e){  
   e.currentTarget.style.backgroundColor = `rgb(${*Math*.floor(*Math*.random() \* 255)},  
   ${*Math*.floor(*Math*.random()\* 255)},  
   ${*Math*.floor(*Math*.random() \* 255)})`;  
    
   *console*.log(this)  
    
  }
* O metoda mai buna e de a pune un event handler la intreaga lista de butoane, adica la parent element. **Asta e mai buna metoda!.** De ex, lista are clasa nav\_\_links si fiecare buton, adica <a> are clasa nav\_\_link, deci am putea extrage href de la <a> ca sa vedem catre ce id are el referinta, caci de ex href=”#jos” ne va duce la elementul cu id ‘jos’
* Ca sa nu se intample asta cand apasa undeva in afara vreunui element, verificam daca nu cumva s-a apasat pe lista, nu pe un element particular

*document*.querySelector(".nav\_\_links").addEventListener('click', (e) => {  
 const target = e.target;  
 const hrefId = target.getAttribute('href');  
 e.preventDefault();  
  
 if(!target.classList.contains('nav\_\_links')){  
 *document*.querySelector(hrefId).scrollIntoView();  
 }  
})

**this vs event.target vs event.currentTarget**

* this este elementul asupra la care se executa metoda din addEventHandler, adica elementul care o apeleaza la momentul dat. Fiecare element parinte apeleaza si el metoda pentru a trata eventul si deci e logic ca this depinde de elementul ce apeleaza metoda
* event.target – elementul care a pornit eventul, indiferent ca metoda deja ajunge sa fie executata de un element parinte.
* event.currentTarget = this

**Oprirea la propagare**

* Putem sa oprim propagarea eventului catre parinti asa:

e.stopPropagation();

function addColor(e){  
 e.currentTarget.style.backgroundColor = `rgb(${*Math*.floor(*Math*.random() \* 255)},  
 ${*Math*.floor(*Math*.random()\* 255)},  
 ${*Math*.floor(*Math*.random() \* 255)})`;  
  
 e.stopPropagation();  
  
}  
  
*document*.getElementById('div1').addEventListener('click', addColor);  
*document*.getElementById('div2').addEventListener('click', addColor);  
*document*.getElementById('a').addEventListener('click', addColor);

* Putem si face ca propagarea sa se faca nu la bubble stage, ci la captuirng stage, si asta va face ca executia sa inceapa nu de la copil la parinti, ci de la parinti la copil, caci asa merge la capturing stage
* Pentru asta, in addEventListener() mai punem si al 3 parametru ca true:
* *document*.getElementById('div1').addEventListener('click', addColor, true);  
  *document*.getElementById('div2').addEventListener('click', addColor, true );  
  *document*.getElementById('a').addEventListener('click', addColor, true );

**Dezactivarea la href=’#’ din <a>**

* Va trebui doar sa dam e.preventDefault() si asa eventul va dezactiva orice comportament default, inclusiv si teleportarea la pagina principala cu #

**DOM Traversing**

* Putem acces childs dintr-un element sau mai bine zis Node in diferite feluri

1. Folosim querySelect() pentru elementul respectic, de ex:

*console*.log(*document*.querySelector('#div1').querySelector('#extern'))

al 2 querySelector() se va efectua doar pe baza la copiii elementului returnat cu primul querySelector()

<div id='div1'>  
 <div id='div2'>  
 <a href='#' id='a'>Click</a>  
 </div>  
</div>  
<div id='extern'></div>

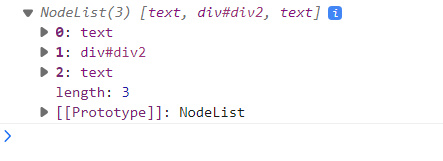
deci, codul de sus va merge, dar daca facem asa ceva:

*console*.log(*document*.querySelector('#div1').querySelector('#extern'))

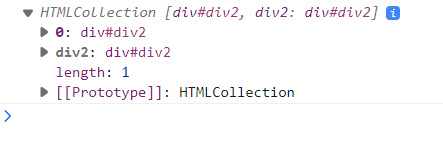
primim null

1. **element.childNodes –** va returna toti copiii ca NodeList

*console*.log(*document*.querySelector('#div1').childNodes)

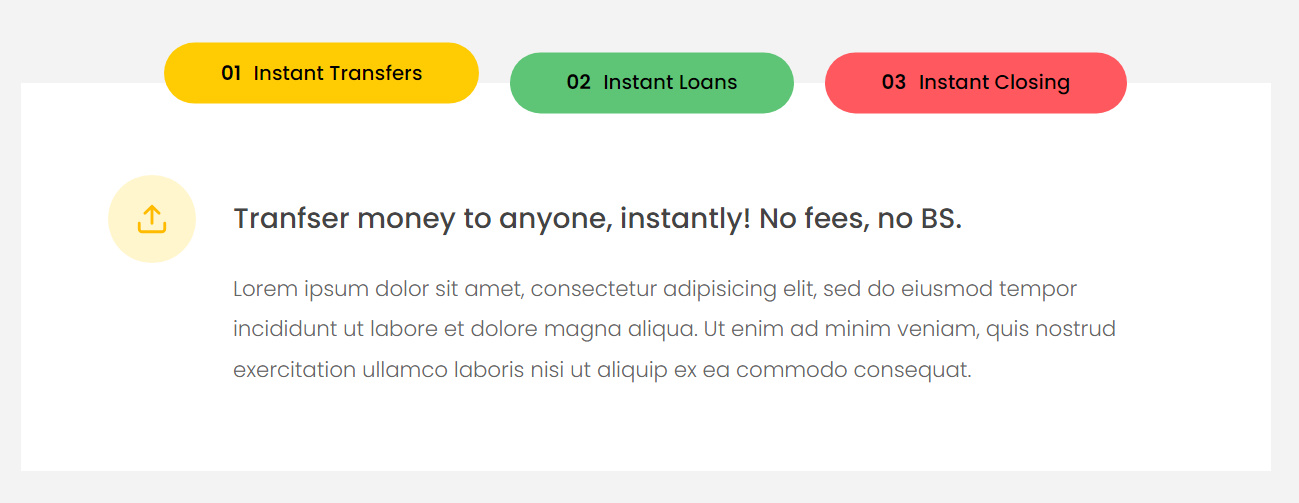


1. **element.children –** returneaza copiii ca HTMLCollection

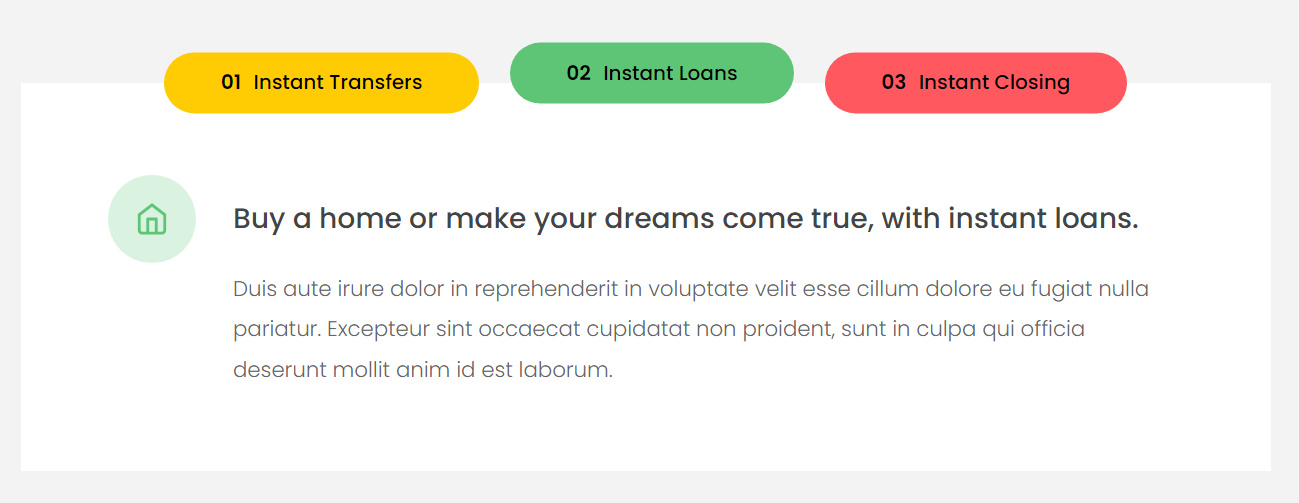
****

* **4. element.first/last|ElementChild**
* Putem obtine si parintii, cu **element.parentElement** si **element.parentNode,** indiferent cat de adanc nu mergem
* Putem obtine si cel mai apropiat element parent, indiferent cat de adanc nu ar fi, cu **element.closest(‘querySelector’).** Poate fi si elementul insusi

**Ex de slider**



apasand pe un buton, el se ridica putin si afiseaza un nou element



Aici apare o problema. Fiecare buton are un numar, gen 01, si asta e un <span> in buton

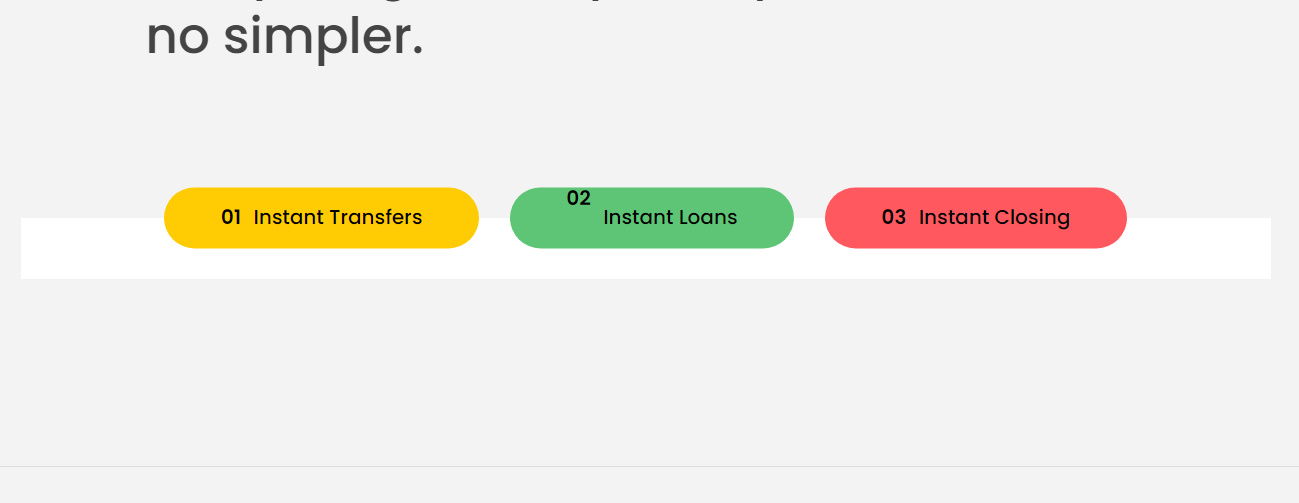
<button class="btn operations\_\_tab operations\_\_tab--3" data-tab="3">  
 <span>03</span>Instant Closing  
</button>

Codul ar fi asa

*tabContainer*.addEventListener('click',(e) =>{  
 const tab = e.target.closest('.operations\_\_tab');  
 const tabs = e.currentTarget.querySelectorAll('.operations\_\_tab');  
  
 if(tab === null) return;  
  
 const slides = *document*.querySelectorAll('.operations\_\_content');  
 const slide = *document*.querySelector(`.operations\_\_content--${tab.dataset.tab}`);  
  
 tabs.forEach(t => t.classList.remove('operations\_\_tab--active'));  
 slides.forEach(s => s.classList.remove('operations\_\_content--active'));  
  
 tab.classList.add('operations\_\_tab--active');  
 slide.classList.add('operations\_\_content--active');  
}

tabelContainer e div unde se afla toate butoanele!

daca nu am folosi closest(), ci doar e.target, si am da click pe acele numere, am pati asta



Problema e ca target nu va fi button, ci <span>, si <span> nu are nimic data-set despre banerul ce-i corespunde. De asta, folosim closest(), ca sa primim mereu un parinte cu clasa .operations\_tab, care e un buton. Orice span are un asa parinte, si chiar de e parintele propriu zis, nicio problema, closest() poate selecta si elementul care o cheama daca el corespunde la query

**Intersection observer API**

* Acest API ne ofera posibilitatea de a crea un subject object, si observers
* Subject object va fi un obiect care va avea mai multi observers, asa si cum prevede Observer pattern
* Ceea ce face anume Intersection observer API este de a notifica toti observers de fiecare data cand subject object se intersecteaza cu ele, si deci executa o anumita functie
* De ex, vrem sa logam in consola un mesaj deficare data cand userul da scroll si ajunge la un anumit element de pe pagina, de ex vre-un <div> specific

const *section1* = *document*.getElementById('section--1');  
const obsCallBack = function(entries, observer){  
 entries.forEach(e => {*console*.log(e.target)});  
}  
const *obsOptions* = {  
root: null,  
threshold: 0,  
rootMargin: '50px'

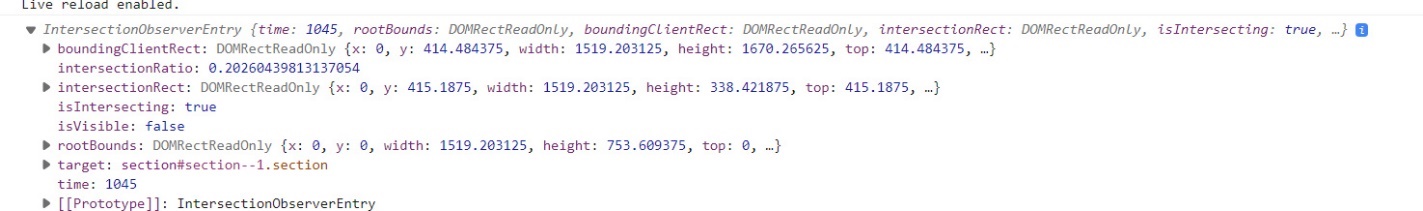
}  
  
const *observer* = new IntersectionObserver(obsCallBack, *obsOptions*);  
*observer*.observe(*section1*);

* Pentru a crea un IntersectionObserver, folosim:

**new IntersectionObserver(function, observer)**

* **function** este functia ce va fi apelata deficare data cand are loc o intersectie

- **entries** – sunt event objects cu care s-a intersectat la un moment dat. Fiecare intersectie e un obiect ce contine informatii, si console.log(e) arata asa:



putem folosi **e.target** pentru a obtine elementul cu care s-a intersectat

- **o** – e obiectul IntersectionObserver propriu zis

const observer = new IntersectionObserver((entries, o) => {

console.log(o === observer)

// True

})

* **obsOptions** e un obiect ce contine fields ca optiuni pentru intersectie

**root**: este obiectul principal, cel ce va verifica cu care alteel el se intersecteaza.

null inseamna ca e window, adica pagina din browser, dar evident, limitata la ce vede userul, nu intreaga pagina

**threshold** – arata cate procente din element trebuie sa se vada ca sa fie trimis eventul la obiectul ce se observa

- .**observer(obj)** – adauga un observer, ca sa verifice daca obiectul setat de noi in root se interstecteaza cu el

**rootMargin** – permite sa marim sau micsoram portview, adica pagina ce e vazuta de user in browser. De ex, rootMargin: ‘50px’ va face ca portview sa fie cu 50px mai mare decat ceea ce noi vedem, atat vertical cat si orizontal. Daca punem ‘-50px’, va fi mai mic cu atata

* etries.isIntersecting

acest field ne arata daca obiectul ce a trecut prin subject abia l-a intersectat ori a iesit din intersectie cu el.

De ex, daca threshold: 0.1, vom primi true la isIntersecting cand elementul subject a intersectat observer in proportie de 10%, si apoi, cand deja iese din intersectia de 10%, adica sa zicem 9.98%, deja va returna false. Eventul e pornit deci si cand obiectele ies din intersectia minima, doar pentru a mai permite sa facem cateva ceva cu ele si in asa situatie.

* e.intersectionRatio

arata procentele de intersectie cand s-a produsul eventul. Daca treshold, e de ex 10%, acest field nu va returna mereu 10 exact, caci scroll se face foarte rapid si adesea pot fi devieri

* **threshold: 1**  inseamna ca elementul trebuie sa fie intersectat in proportie de 100%
* **threshold: 0** (default) inseaman ca se va declansa un event, cu isInterstecting = true cand se vede macar si un pixel din element,o bucata oricat de mica, si apoi events cu isIntersecting = false doar dupa ce elementele vor iesi din intersectie complet
* **threshold** poate primi si un array cu procente, gen [0, 0.25, 0.75] si atunci se va arunca cate un event de fiecare data cand un procentaj e atins si cand el nu mai e atins
* **.unobserve(el)** – scoate elementul ca observer

[JavaScript Intersection Observer Ultimate Guide (webdevsimplified.com)](https://blog.webdevsimplified.com/2022-01/intersection-observer/)

**Incarcarea bucatilor de pagini lazy la scroll**

* Trebuie intai sa facem toate elementele ce nu sunt in viewpoer vizibile ascunse, apoi le vom afisa cu transform:
* Folosim InterstectionObserver si de fiecare data cand cu scroll ne apropiem de vreun element, il afisam

const *sectionObserverOptions* = {  
 root: null,  
 threshold: 0.15  
}  
  
const *sectionObserver* = new IntersectionObserver((elements) => {  
 elements.forEach((e) => {  
 if(e.isIntersecting) {  
 e.target.classList.remove('section--hidden');  
 *sectionObserver*.unobserve(e.target);  
 }  
 })  
})  
  
*document*.querySelectorAll('.section').forEach(e => {  
 e.classList.add('section--hidden');  
 *sectionObserver*.observe(e)  
});

odata ce un element e afisat pe pagina, putem sa il scoatem din lisat de observers, asa cum nu mai are sens sa mai fie incarcat din nou

**Lazy loading images**

* Imaginile au un mare impact asupra vitezei de incarcare a unei pagini
* Iata de ce, e mai bine sa fie lazy
* Prin lazy, inseamna ca isneram o imagine mica, cu o calitate foarte mica, si cand userul ajunge la ea, o inlocuim cu cea cu calitate mare
* O putem face sa fie si cu blur, ca sa nu aratam cat de urata e
* Pentru a specifica imaginea cu calitate buna, putem folosi data attribute:

<img  
 src="img/digital-lazy.jpg"  
 **data-src="img/digital.jpg"**  
 alt="Computer"  
 class="features\_\_img lazy-img"  
/>

const *imgObserverConfig* = {  
 root: null,  
 threshold: 0,  
 rootMargin: '100px'  
}  
const *imgObserver* = new IntersectionObserver((entries) => {  
 entries.forEach(e => {  
 if(e.isIntersecting) {  
 e.target.src = e.target.dataset.src;  
 e.target.classList.remove('lazy-img');  
 *imgObserver*.unobserve(e.target);

}  
 }, *imgObserverConfig*)  
})  
  
*document*.querySelectorAll(".lazy-img").forEach((img) => {  
 *imgObserver*.observe(img);  
})

Totusi, mai e o mica problema, si anume ca clasa lazy-img va disparea deodata, posibil pana imaginea sa se incarce, si daca viteza la net e slaba, pana se va incarca imaginea buna, imaginea urata sau de calitate joasa inca va ramane, fara blur dejaz

const *imgObserver* = new IntersectionObserver((entries) => {  
 entries.forEach(e => {  
 if(e.isIntersecting) {  
 e.target.src = e.target.dataset.src;  
 e.target.addEventListener('load',(el) => el.target.classList.remove('lazy-img'));  
 *imgObserver*.unobserve(e.target);  
 }  
 }, *imgObserverConfig*)  
})

asa, acum clasa care punea blur va fi scoasa doar cand imaginea va fi incarcata, caci asta si inseamna ‘load’ event, cand obiectul e incarcat complet.

**Slider Component**

* 
* let *currentSlide* = 0;  
    
  const *slideButtonLeft* = *document*.querySelector('.slider\_\_btn--left');  
  const *slideButtonRight* = *document*.querySelector('.slider\_\_btn--right')  
  const *slides* = new *Array*(...*document*.querySelectorAll('.slide'));  
    
  *slides*.forEach((e, index) => {  
   e.style.transform = `translateX(${100\*index}%)`;  
  })  
    
  function changeSlide(){  
   *slides*.forEach((slide, index) => {  
   let size = (index - *currentSlide*)\*100;  
   slide.style.transform = `translate(${size}%)`;  
   })  
  }  
  function changeSlideRight(e) {  
   if (*currentSlide* === *slides*.length - 1)  
   *currentSlide* = 0;  
   else  
   *currentSlide*++;  
    
   changeSlide();  
  }  
  function changeSlideLeft(e){  
   if (*currentSlide* === 0)  
   *currentSlide* = *slides*.length-1;  
   else  
   *currentSlide*--;  
    
   changeSlide();  
  }  
  *slideButtonRight*.addEventListener('click', changeSlideRight);  
  *slideButtonLeft*.addEventListener('click', changeSlideLeft)

**DOM Events**

* **DOMContentLoaded** – e aruncat cand pagina HTML a fost incarcata si scripturile la fel si a fost creat DOM Tree. Nu asteapta ca imaginile sau alte surse externe sa fie incarcate, doar JS si HTML
* **load** – e aruncat cand HTML, JS, imgs si resursele externe sunt incarcate!
* **beforeunload –** e aruncat inainte ca userul sa paraseasca pagina, adica cand apasa pe close si apare un avertisment de confirmare

**Script Loading**

* **Regular:**

- **Head**: browser va incepe sa incarce codul HTML, si la un moment dat, va incepe sa descarce si execute scriptul JS, iar in acest timp, va opri procesul de incarcare a codului HTML si a DOM Tree. Asta e rau, caci scriptul poate fi executat inainte ca toata pagina HTML sa fie incarcata si DOM TREE creat si asta inca va influenta performanta, asa cum browserul nu va mai incarca pagina HTML ci va astepta

- **Body End** – cea mai buna varianta. HTML e inarcat tot, DOM Tree e creati si deja apoi se executa scriptul

* **Async**: script e descarcat in acelasi timp cu HTML. Totusi, odata de script e descarcat, el se executa, si in acest timp, HTML iar nu mai e incarcat si asteapta
* **Defer(**  – script e descarcat in acelasi timp cu HTML, insa executia JS incepe abea cand HTML e incarcat complet

Nu are sens sa punem Async sau Defer la final de body, caci incarcarea lor oricum va incepe mereu deja dupa ce HTML a fost descarcat.

