ChatGPT

JavaScript Starter Pack (Osnove JavaScript-a)

Uvod u JavaScript

JavaScript je **skriptni programski jezik** koji omogućava dodavanje interaktivnosti i logike web stranicama. Za razliku od HTML-a i CSS-a (koji opisuju strukturu i stil stranice), JavaScript omogućava *programsku kontrolu* – reagovanje na korisničke akcije, manipulaciju sadržaja na stranici, validaciju unosa, komunikaciju sa serverom i sl. Pokreće se prvenstveno u veb pregledaču na klijentskoj strani (svaki moderni browser ima ugrađen JavaScript pokretač), ali se danas koristi i van brauzera (npr. **Node.js** za serversko izvođenje JavaScript koda).

Karakteristike: JavaScript je **dinamički tipiziran** jezik visokog nivoa, što znači da pri deklaraciji promenljive ne navodimo tip podatka, a jedna promenljiva tokom izvršavanja programa može sadržati različite tipove vrednosti. Interpretiran je (izvodi se *runtime* u pregledaču ili uz pomoć okruženja kao Node.js, bez prethodne eksplicitne kompilacije). Takođe je višestruko paradigma – podržava objektno-orijentisani stil, funkcionalno programiranje, imperativno programiranje i dr. Sintaksu je delimično nasledio od C/Java jezika (npr. način pisanja petlji, uslova, korišćenje viticastih zagrada `{}` itd.), pa će neke konstrukcije biti prepoznatljive ako imate iskustva sa tim jezicima [1] .

Gde se koristi: JavaScript je standardni jezik za razvoj *front-end* veb aplikacija – praktično svaka savremena web stranica koristi JavaScript za interakciju (od prostog prikaza aktuelnog datuma na stranici, do kompleksnih web aplikacija). U praksi, to znači da se JS koristi za validaciju formi u pregledaču, prikaz dinamičkog sadržaja (npr. ažuriranje dela stranice bez ponovnog učitavanja cele stranice - tzv. AJAX pozivi ka serveru), animacije, prikaz notifikacija, izgradnju bogatih korisničkih interfejsa i još mnogo toga. Pored *front-enda*, uz Node.js JavaScript se koristi i na *back-endu* (serveri, komande linije alati), u hibridnim mobilnim aplikacijama, skriptovanju i dr. To znači da je JavaScript danas univerzalni jezik koji vredi temeljno naučiti.

Kako pokrenuti JavaScript kod: U kontekstu web stranice, JavaScript se najčešće ubacuje u HTML dokument pomoću `<script>` tagova. Primer – u HTML fajlu možete dodati:

```
<script>
  console.log("Pozdrav iz JavaScripta!");
  alert("Hello, world!");
</script>
```

Ovaj kod će se izvršiti kada se stranica učita – prikazaće poruku u konzoli pregledača i iskačući prozor (*alert*). Alternativno, možete JavaScript smestiti u poseban fajl (npr. *script.js*) i uključiti ga u HTML pomoću `<script src="script.js"></script>`. Još jedan način za brzo testiranje JS koda je korišćenje **konzole pregledača** – pritisnite F12 (ili desni klik pa “Inspect/Inspect Element” -> Console) i u konzoli možete kucati JavaScript izraze i odmah videti rezultat. Ovo je odlično za isprobavanje jednostavnih naredbi i razumevanje kako jezik funkcioniše.

> **Napomena:** JavaScript **nije** isto što i Java – uprkos sličnom imenu, to su potpuno različiti jezici. JavaScript je dobio ime u vreme popularnosti Jave, ali sintaksom i primenom više

podseća na jezike poput Python-a ili Perl-a. Takođe, moderni JavaScript (ES6 i novije verzije) doneo je mnoga unapređenja jezika (npr. *let/const* promenljive, *arrow* funkcije, klase, Promise/async, itd.), pa ćemo uvek akcenat stavljati na **savremene najbolje prakse**.

Promenljive i tipovi podataka

Promenljive služe za skladištenje podataka pod određenim imenom radi kasnijeg korišćenja u programu. U JavaScript-u se promenljiva može proglasiti ključnom reči `let`, `const` ili (starijom) `var`. Preporuka je da u modernom JS gotovo uvek koristite `let` i `const` umesto `var` [2]. Razlike su sledeće:

- `var` je istorijski način deklaracije; promeljive definisane sa `var` imaju ili *globalni* ili *funkcijski* opseg (scope). Mogu se *ponovo deklarisati* u istom opsegu i dozvoljeno im je naknadno dodeljivanje druge vrednosti.
- `let` je uveden u ES6 (2015) i ima **blokovski opseg** – važi samo unutar bloka koda `{ ... }` gde je definisan [3]. `let` promenljiva **može menjati vrednost** (nije konstantna) ali **ne može biti ponovo deklarisana** unutar istog opsega [4] [5].
- `const` takođe ima blokovski opseg i koristi se za **konstante** – vrednosti koje se ne menjaju. *Mora* se inicijalizovati prilikom deklaracije i **nije dozvoljena promena** vrednosti niti ponovna deklaracija [6] [7]. Ipak, za složene tipove (objekte, nizove) *const* sprečava promenu referencije, ali ne i izmene unutrašnjeg sadržaja objekta/niza [8] [9].

U praksi, pravilo je: koristite `const` kad god je moguće (za vrednosti koje neće menjati tokom izvršavanja), a `let` za promenljive čija će se vrednost menjati. `var` se u modernom JavaScript-u uglavnom više ne koristi (može dovesti do neintuitivnog ponašanja zbog *hoistinga* i šireg opsega delovanja nego što se očekuje). Sažetak razlika: `var` **je funkcijski ili globalni,** `let` **/** `const` **su blokovski opseg;** `var` **dozvoljava ponovnu deklaraciju,** `let` **ne,** `const` **ne dozvoljava ni promenu vrednosti; pri hoistingu** `var` **dobija vrednost** `undefined`**, dok** `let/const` **ne (što znači da ih ne možete koristiti pre deklaracije)** [2].

Tipovi podataka

JavaScript ima **osam osnovnih tipova podataka** (sedam *primitivnih* i jedan kompleksni) prema ECMAScript standardu [10] [11]:

1. **Brojevi** (`Number`) – predstavljaju numeričke vrednosti (celobrojne ili realne). U JS svi brojevi su 64-bitni *floating-point* (IEEE 754 standard), npr. `let x = 42;`, `let y = 3.14;`. Posebne vrednosti: `Infinity`, `-Infinity` i `NaN` (Not-a-Number, rezultat nedozvoljenih aritmetičkih operacija). Za celobrojne vrednosti veće od 2⁵³–1 koristi se posebna vrsta: **BigInt**.
2. **Veliki integer** (`BigInt`) – predstavlja celobrojne vrednosti arbitrarne veličine. Dodaje se sufiks `n` na broj, npr. `let big = 9007199254740993n;`. BigInt omogućava rad sa veoma velikim celinama koje `Number` ne može precizno reprezentovati.
3. **String (niz karaktera)** – tekstualne vrednosti u navodnicima. Može se koristiti jednostruki `'tekst'`, dvostruki `"tekst"` ili obrnuti apostrof <code>`tekst`</code> (tzv. *template literal*). Primeri: `let s1 = "Hello"; let s2 = 'World';`. *Template literals* (označeni sa <code>`</code>) omogućavaju interpolaciju izraza – npr:

```
let ime = "Mika";
console.log(`Zdravo, ${ime}!`); // ispisuje: Zdravo, Mika!
```

Takođe, **string** ima svojstvo `.length` (dužina niza) i podržava mnoge metode (npr. `toUpperCase()`, `includes()` i dr.).

4. **Boolean (logički)** – dve moguće vrednosti: `true` ili `false`. Koristi se za logiku, poređenja i uslovne izraze.
5. `null` – posebna ključna reč koja predstavlja *praznu* ili nepostojeću vrednost. To je eksplicitna "ništa" vrednost koju programer dodeljuje promenljivoj da označi odsustvo vrednosti.
6. `undefined` – označava da promenljivoj nije dodeljena vrednost. Svaka deklarisana promenljiva koja nema inicijalnu vrednost biće `undefined` podrazumevano. Takođe, ako funkcija nema `return` naredbu, vraća `undefined`. (Razlika `null` vs `undefined`: `null` je namerno postavljena "ništa" vrednost, dok `undefined` znači da vrednost uopšte nije definisana).
7. **Symbol (simbol)** – noviji primitivan tip (ES6) koji predstavlja jedinstvenu, anonimnu vrednost. Koristi se uglavnom za posebne slučajeve, poput definisanja jedinstvenih ključeva u objektima koji neće konflikti sa drugim. Kreira se pozivom `Symbol()`.
8. **Object (objekat)** – kompleksni tip koji obuhvata niz vrednosti grupisanih po ključu (key-value parovi) ili pak funkcionalnost. Svi preostali tipovi koji nisu primitivni jesu objekti. Npr. nizovi (Array), funkcije, datumi, regularni izrazi, itd. su poseban podtip objekata. Objekti zaslužuju posebno poglavlje (videti dole **Objekti**), jer su centralni koncept JS-a.

JavaScript je **dinamički tipiziran** jezik – ne navodimo tip prilikom deklaracije promenljive, a jedna promenljiva može tokom rada programa sadržati vrednosti različitih tipova. Na primer:

```
let odgovor = 42;          // broj
odgovor = "Zdravo";        // sada je promenljiva string
console.log(odgovor);      // ispisuje "Zdravo"
```

Ovo je dozvoljeno jer JavaScript **automatski konvertuje tipove po potrebi** i ne veže promenljivu za jedan tip [12] [13]. Iako fleksibilno, ovo zahteva pažnju – moguće su implicitne konverzije koje ponekad daju neočekivane rezultate (videti deo o operatorima, npr. dodavanje broja i stringa).

> **Napomena:** Da biste saznali tip neke vrednosti u JavaScript-u, možete koristiti operator `typeof`. Npr: `typeof 5` vraća `"number"`, `typeof "hello"` vraća `"string"`, dok za null i nizove postoje specifičnosti: `typeof null` vraća `"object"` (što je bag u jeziku koji je ostao iz istorijskih razloga), a `typeof [1,2,3]` takođe vraća `"object"` (jer su nizovi implementirani kao objekti). Prepoznavanje da je nešto niz može se uraditi sa `Array.isArray(...)`.

Operatori i izrazi

Operatori su simboli koji vrše određene operacije nad operandima (vrednostima ili promenljivima). Rezultat evaluacije operatora i operanada naziva se **izraz**. JavaScript podržava standardne aritmetičke, logičke i druge operatore slične drugim programskim jezicima [14]. Pregled najvažnijih:

- **Aritmetički operatori:** `+` (sabiranje), `-` (oduzimanje), `*` (množenje), `/` (deljenje), `%` (modulo – ostatak pri deljenju), `**` (stepenovanje, npr. `2 ** 3` iznosi 8). **Napomena:** operator `+` ima dvojaku ulogu – osim sabiranja brojeva, koristi se i za **konkatenaciju stringova**. Npr:

`"Hello, " + "world!"` daje `"Hello, world!"`. Ako kombinujete broj i string sa `+`, broj će se prebaciti u string: `"Score: " + 100` rezultuje `"Score: 100"` [15]. Sa ostalim aritmetičkim operatorima, string koji sadrži broj će se pokušati konvertovati u broj: `"37" - 7` daje `30`, `"37" * 7` daje `259` [16] [17].

- **Operator dodele (assignment):** znak jednakosti `=` koristi se za dodelu vrednosti promenljivoj. *Važno:* ovo **nije** poređenje već dodeljivanje! Primer: `let a = 5;` (dodeljuje 5 u a). Postoje i **kombinovani operatori dodele**: `+=`, `-=`, `*=`, `/=`, npr. `x += 3;` je skraćeno `x = x + 3;`.
- **Uporedni (relacioni) operatori:** koriste se za poređenje vrednosti i vraćaju Boolean rezultat. Najčešći: `==` jednako (loose equality), `!=` nije jednako, `===` striktno jednako, `!==` striktno nije jednako, `>` veće, `<` manje, `>=` veće ili jednako, `<=` manje ili jednako. *Preporuka:* Uvek koristite **strogo jednako** `===` i **strogo različito** `!==` za poređenje, jer **dvodupljednak** `==` radi implicitnu konverziju tipova što može dovesti do neočekivanih rezultata. Ukratko: `==` će pokušati da konvertuje operande na isti tip pre poređenja (npr. `"5" == 5` je `true` jer string "5" se konvertuje u broj 5), dok `===` poredi *i vrednost i tip* bez konverzije (npr. `"5" === 5` je `false` jer su različitog tipa) [18]. Zbog toga je `===` **bezbedniji** – ako poređenje uključuje različite tipove, odmah će dati `false` umesto da pokušava konverziju. Isto važi i za `!==` naspram `!=`.
- **Logički operatori:** `&&` (logičko **I**), `||` (logičko **ILI**), `!` (logička **NE** negacija). Ovi operatori rade sa Bool vrednostima (ili vrednostima koje mogu biti tretirane kao true/false – vidi *truthy/falsy* napomenu ispod). Primena: uslovne provere, kombinovanje više uslova itd. Npr: `(x > 0 && x < 10)` je true ako je x između 0 i 10. `!flag` je true ako je flag false.
- **Ternarni operator:** `? :` – jedini uslovni operator koji ima tri operanda. Sintaksa: `uslov ? vrednost_if_true : vrednost_if_false`. Evaluira uslov, ako je istinit vrati prvu vrednost, ako je lažan vrati drugu. Primer: `let poruka = isMember ? "Dobrodošli" : "Registrujte se";`. Ovaj izraz dodeljuje jednu od dve string vrednosti promenljivoj *poruka* zavisno od bool promenljive *isMember*.

Izrazi su kombinacije vrednosti, promenljivih i operatora koje JavaScript može izračunati. Npr. `3 + 4` je aritmetički izraz koji se evaluira u 7. Funkcije takođe mogu biti deo izraza, npr. `Math.sqrt(x) + 5`. Postoje i posebni operatori poput `typeof` (opisano gore) i `delete` (za brisanje svojstva objekta), ali njih ćemo obraditi kasnije u kontekstu objekata.

Istinitost (truthy/falsy): U JavaScript-u **svaka vrednost** se može tretirati kao boolean u kontekstu uslova. Vrednosti koje se smatraju "lažnim" (falsy) su: `false`, `0`, `0n` (BigInt nula), `""` (prazan string), `null`, `undefined` i `NaN`. Sve ostalo se smatra "istinitim" (truthy), uključujući i ne-prazne stringove, sve objekte, nizove, pa čak i `"0"` (string koji sadrži nulu) ili `[]` (prazan niz) su truthy. Ovo je bitno kod logičkih operatora i if uslova – npr. `if (x) {...}` će ući u blok ako je x truthy. Takođe, logički operatori `&&` i `||` u JS vraćaju jednu od vrednosti umesto isključivo true/false: `a && b` vraća `a` ako je falsy, inače vraća `b`; `a || b` vraća `a` ako je truthy, u suprotnom vraća `b`. Ovo se često koristi za dodelu podrazumevanih vrednosti, npr. `let ime = inputName || "Gost";` (ako je *inputName* prazan/falsy, uzmi "Gost").

Uslovne naredbe (if / else i switch)

Uslovni iskazi omogućavaju grananje toka programa na osnovu ispunjenosti nekog uslova (logičkog izraza). Najvažnija je `if` naredba, često kombinovana sa `else`:

```
let vreme = 20;
if (vreme < 12) {
  console.log("Dobro jutro!");
} else if (vreme < 18) {
  console.log("Dobar dan!");
} else {
  console.log("Dobro veče!");
}
```

U ovom primeru, ispis zavisi od vrednosti promenljive *vreme*. Prvo `if` proverava uslov (da li je vreme manje od 12). Ako je uslov tačan, izvršiće se blok koda u vitičastim zagradama odmah ispod njega, a ostali `else if/else` delovi se preskaču. Ako uslov nije bio tačan, proverava se naredni `else if`, itd. `else` blok (koji dolazi na kraju) izvršiće se jedino ako nijedan prethodni uslov nije bio ispunjen. `else if` delovi su opcioni i može ih biti više, dok je `else` takođe opcioni deo za slučaj "svega ostalog".

Switch je druga forma grananja koja je pogodna kada se isti izraz poredi sa više različitih vrednosti. Sintaksa:

```
let dan = 3;
switch (dan) {
  case 1:
    console.log("Ponedeljak");
    break;
  case 2:
    console.log("Utorak");
    break;
  case 3:
    console.log("Sreda");
    break;
  default:
    console.log("Neki drugi dan");
}
```

U *switch* naredbi, izraz (ovde *dan*) se poredi sa navedenim **case** konstantama. Kod prvog case koji se podudari, izvrši se odgovarajući blok koda. `break` je važan – on prekida dalje izvođenje switch-a, inače bi program nastavio da izvršava i naredne case-ove (*fall-through* ponašanje). `default` case (opcion) hvata slučaj kada nijedan konkretan case nije zadovoljio – slično else grani.

Switch je najčešće koristan kada imate fiksne vrednosti za poređenje. Za složenije uslove bolje je koristiti if/else. Takođe, uslovni (ternarni) operator `? :` može zameniti prost if/else u situacijama dodele vrednosti (vidi gore), ali za duže blokove logike if/else je pregledniji.

Petlje (naredbe ponavljanja)

Petlje služe za ponavljanje izvršavanja nekog bloka koda dok god važi određeni uslov, ili za iteraciju kroz kolekcije podataka. JavaScript podržava nekoliko vrsta petlji:

- `for` **petlja (brojačka petlja):** koristi se kad unapred znamo koliko puta treba izvršiti korak ili iterirati kroz niz brojeva. Sintaksa: `for (inicijalizacija; uslov; ažuriranje) { ... }`. Primer:

```
for (let i = 1; i <= 5; i++) {
  console.log("Iteracija broj " + i);
}
```

Ova petlja će ispisati `Iteracija broj 1` do `Iteracija broj 5`. Mehanizam: prvo se izvrši *inicijalizacija* (`let i = 1`), zatim pre svake iteracije proveri *uslov* (`i <= 5`); ako je uslov true, uđe u petlju i izvrši telo, pa onda uradi *ažuriranje* (`i++`, povećanje brojača) i ponovo proveri uslov... sve dok uslov ne postane false.

- `while` **petlja:** ponavlja blok koda **dok** je neki uslov ispunjen. Primer:

```
let k = 1;
while (k <= 5) {
  console.log(`K = ${k}`);
  k++;
}
```

Ova petlja radi slično prethodnom for primeru – dok je `k <= 5` ispisuje vrednost k i uvećava ga. Razlika je sintaksna: `while` nema ugrađenu inicijalizaciju i update, to se obavlja unutar petlje ili pre nje. Ako se uslov nikad ne promeni u false, došlo bi do *beskonačne petlje* – zato pazite da u telu petlje menjate varijablu koja utiče na uslov, inače će program "zaglaviti" u toj petlji.

- `do...while` **petlja:** slična while petlji, ali proverava uslov *nakon* izvršavanja tela, što garantuje da će telo petlje biti izvršeno bar jednom. Sintaksa:

```
let password;
do {
  password = prompt("Unesite ispravnu lozinku:");
} while (password !== "12345");
```

U ovom primeru, *prompt* za unos lozinke pojaviće se barem jednom, pa tek onda proveravamo uslov (da li lozinka nije "12345"). Ako uslov nije ispunjen, petlja se ponavlja.

- `for...of` **petlja:** uvedena u ES6, služi za iteraciju kroz elemente **iterabilnih** objekata (poput nizova, stringova, mapa...). Primer sa nizom:

```
let brojevi = [10, 20, 30];
for (let br of brojevi) {
  console.log(br);  // ispisuje 10, zatim 20, zatim 30
}
```

Ova petlja prolazi kroz niz i u svakoj iteraciji promenljiva *br* dobije naredni element niza. `for...of` je često najjednostavniji način da prođete kroz sve vrednosti kolekcije. (Napomena: postoji i `for...in`, ali on se koristi za iteraciju kroz **ključeve** objekta – ređe se primenjuje za nizove jer prolazi i preko *prototipskih* svojstava; ukratko, `for...in` koristite za objekte, a `for...of` za nizove i iterabile).

Tokom petlji možemo koristiti `break` i `continue` naredbe za kontrolu toka: - `break` – trenutni prekid petlje, izlazak iz nje (najčešće unutar if uslova kada je postignut cilj). - `continue` – preskače preostale naredbe u telu za tu iteraciju i prelazi odmah na sledeću iteraciju petlje (odnosno, radi skok na ažuriranje i ponovnu proveru uslova u for petlji, ili direktno proveru uslova kod while).

Primer:

```
for (let n = 1; n <= 10; n++) {
  if (n % 2 === 0) continue; // preskoči parne brojeve
  if (n > 7) break;          // prekini petlju kad n pređe 7
  console.log(n);            // ispisuje 1, 3, 5, 7
}
```

Funkcije

Funkcije su osnovni blokovi za organizaciju koda. Omogućavaju grupisanje skupa naredbi koje izvršavaju određeni zadatak, a zatim se mogu višekratno pozivati po imenu. Korišćenje funkcija sprečava ponavljanje koda (princip *DRY*: Don't Repeat Yourself) i poboljšava modularnost i čitljivost programa.

Deklaracija funkcije: Ključna reč `function` uvodi deklaraciju funkcije, zatim ide ime funkcije, lista parametara u zagradama, i telo funkcije u vitičastim zagradama. Na primer:

```
function zbir(a, b) {
  let rezultat = a + b;
  return rezultat;
}
```

Ova funkcija `zbir` prihvata dva parametra (*a* i *b*), sabira ih i vraća rezultat. `return` naredba prekida izvršavanje funkcije i opcionalno *vraća vrednost* pozivaocu. Ako se `return` izostavi ili nema eksplicitne vrednosti, funkcija vraća `undefined` podrazumevano.

Funkciju pozivamo njenim imenom i prosleđivanjem argumenata: npr. `let x = zbir(5, 7);` – nakon ovog poziva, promenljiva *x* ima vrednost 12. Funkcija može imati i **0 parametara** (npr. `function pozdrav(){ console.log("Ćao"); }`), a može i vratiti ništa (npr. samo izvršavati akcije,

tzv. *procedura*). Parametri koje funkcija primi tretiraju se kao lokalne promenljive unutar tela funkcije. JavaScript ne zahteva deklarisanje tipova parametara niti povratne vrednosti.

Funkcije su građani prvog reda: Možemo ih tretirati kao vrednosti. To znači da funkciju možemo dodeliti promenljivoj, prosleđivati je kao argument drugoj funkciji ili vratiti iz funkcije. Primer *funkcionalnog* pristupa – dodela anonimne funkcije promenljivoj:

```
const kvadrat = function(n) { return n * n; };  // funkcijski izraz, nema ime
console.log( kvadrat(4) );  // 16
```

Ovde smo kreirali funkciju *bez imena* (anonimnu) i dodelili je konstanti *kvadrat*. Sada se *kvadrat* ponaša kao funkcija koju možemo pozvati. Funkcijski izrazi mogu biti korisni za **callback** funkcije (funkcije koje se prosleđuju kao argument i pozivaju unutar druge funkcije, npr. u event handlerima ili metodama poput `array.map()`).

Streličaste funkcije (arrow functions): ES6 je uveo kraći sintaksni zapis za funkcije pomoću tzv. "streličaste" sintakse `=>`. Arrow funkcije omogućavaju da brže definišemo funkciju posebno kada je jednostavna. Primer koji funkcijski izraz iznad zapisuje kao arrow:

```
const kvadrat = (n) => n * n;
```

Ovo je ekvivalentno prethodnom kodu – jednolinijska arrow funkcija sa jednim parametrom *n* i povratnom vrednošću `n * n` (ovde nije potreban `return` ni vitičaste zagrade jer se telo sastoji od jedne izrazne linije). Ako arrow funkcija ima jedan parametar, mogu se izostaviti i obične zagrade oko parametra [19] [20]. Ako nema nijedan parametar, stavlja se prazne zagrade `()`. Za više linija koda u telu arrow funkcije, koristimo vitičaste zagrade i eksplicitno `return` za povratnu vrednost.

Arrow funkcije su *leksički određene* u pogledu konteksta izvršavanja, što znači da **nemaju sopstveni** `this` – uzimaju `this` vrednost iz okruženja u kom su definisane [21]. Ovo ih razlikuje od regularnih funkcija i može biti prednost u situacijama kada radimo sa objektima i event handlerima (eliminisana je česta potreba za hvatanjem `this` u promenljivu). U praksi, arrow funkcije su idealne za kratke callbacke i funkcije koje ne zahtevaju sopstveni `this` ili ne koriste `arguments` objekat. Za metode objekata i složenije funkcije, klasična deklaracija je i dalje sasvim u redu.

Primer: nekoliko načina definisanja iste funkcije:

```
// 1. Deklaracija funkcije
function pozdrav(ime) {
  console.log("Zdravo, " + ime + "!");
}

// 2. Funkcijski izraz (anonimna funkcija dodeljena promenljivoj)
const pozdrav2 = function(ime) {
  console.log("Zdravo, " + ime + "!");
};

// 3. Arrow funkcija ekvivalentna gorenavedenom
```

```
const pozdrav3 = (ime) => {
  console.log(`Zdravo, ${ime}!`);
};
```

Sve tri funkcije iznad ponašaju se isto (`pozdrav("Ana")` ispisuje "Zdravo, Ana!", i isto tako `pozdrav2("Ana")` i `pozdrav3("Ana")`). Važno je napomenuti da klasične funkcije definisane *deklarativno* (`function ime() {...}`) trpe hoisting – mogu se pozvati čak i pre nego što su definisane u kodu (interpretator ih interno podigne na vrh). Nasuprot tome, funkcije definisane kao izraz (bilo klasične ili arrow, dodeljene promenljivoj) **ne mogu** se pozvati pre njihove definicije u kodu, jer se ponašaju poput bilo koje promenljive koja mora prvo biti inicijalizovana.

Nizovi (Arrays)

Niz (array) je strukturirani tip koji predstavlja uređenu listu vrednosti. Nizovi su pogodni za čuvanje kolekcije elemenata (brojeva, stringova, objekata itd.) koje možemo adresirati putem indeksa. U JavaScript-u se nizovi zapisuju u **ugaonim zagradama** `[...]`, a elementi su odvojeni zarezima:

```
let praznaLista = [];                  // prazan niz
let brojevi = [10, 20, 30];            // niz brojeva
let razno = [1, "dva", true, null];    // niz različitih tipova
(dozvoljeno)
```

Niz može sadržati elemente različitih tipova (dynamically typed nature), pa čak i druge nizove (tada imamo *multidimenzionalne* nizove). **Indeksiranje:** elementi niza su indeksirani celim brojevima od 0 nadalje. Dakle, u nizu `brojevi` iz primera, `brojevi[0]` je 10, `brojevi[1]` je 20 itd. Možemo i menjati vrednosti na odreenim pozicijama: npr. `brojevi[1] = 25;` promeni drugi element niza u 25. Dužina niza je dostupna kroz svojstvo `.length` (npr. `brojevi.length` je 3). **Napomena:** Indeks poslednjeg elementa je `length-1`. Ako pokušate pristup indeksa koji ne postoji (>= length), dobićete `undefined`.

Iteracija kroz niz: Najjednostavnije je pomoću klasične for petlje ili `for...of` petlje. Npr:

```
for (let i = 0; i < brojevi.length; i++) {
  console.log(brojevi[i]);
}
// ili ES6 način:
for (let vrednost of brojevi) {
  console.log(vrednost);
}
```

Mutiranje niza: Nizovi su u JavaScript-u **objekti** specijalne vrste (imaju prototip Array sa mnoštvom ugrađenih metoda). Neki od često korišćenih *metoda za manipulaciju nizova*:

- `push(elem)` – dodaje novi element *na kraj* niza.
- `pop()` – uklanja poslednji element niza i vraća ga.
- `shift()` – uklanja **prvi** element niza (pomera ostale ulevo).
- `unshift(elem)` – dodaje novi element na početak niza (pomera postojeće udesno).

- `slice(start, end)` – vraća **novi** niz koji je podskup originalnog (od indeksa start do end-1).
- `splice(start, deleteCount, elem1, elem2, ...)` – **menja originalni niz**: može brisati elemente (deleteCount broj njih počev od start indeksa) i/ili ubacivati nove na njihovo mesto.
- `indexOf(vrednost)` – vraća indeks prve pojave *vrednost* u nizu (ili -1 ako nije nađen).
- `includes(vrednost)` – vraća true/false zavisno od toga da li se data vrednost nalazi negde u nizu.

Primer manipulacije niza:

```
let list = ["jabuka", "banana"];
list.push("citrus");               // ["jabuka", "banana", "citrus"]
list.unshift("kivi");              // ["kivi", "jabuka", "banana", "citrus"]
list.pop();                        // uklanja "citrus", sada ["kivi",
"jabuka", "banana"]
console.log(list[1]);              // "jabuka"
console.log( list.includes("kivi") ); // true
```

Nizovi takođe imaju korisne **iterativne metode**: `forEach` (poziva zadatu funkciju za svaki element), `map` (kreira novi niz transformacijom svakog elementa), `filter` (filtrira elemente po uslovu), `find` (pronalazi prvi element koji ispunjava uslov) itd. Te metode često olakšavaju rad sa nizovima na deklarativan način umesto eksplicitnih for petlji. Npr:

```
let brojevi2 = [1, 2, 3, 4];
let kvadrati = brojevi2.map(x => x * x);      // [1, 4, 9, 16]
let parni = brojevi2.filter(x => x % 2 === 0); // [2, 4]
```

Za početak, važno je razumeti osnovne principe rada s nizovima (indeksiranje, length, dodavanje/uklanjanje), a vremenom ćete prirodno usvojiti i bogat skup dostupnih metoda.

Objekti

Objekat je kolekcija svojstava (osobina) gde je svako svojstvo definisano parom **ključ: vrednost**. Ključevi (propiedades) su tipično stringovi (ili *Symbol*), dok vrednosti mogu biti bilo kog tipa – uključujući i druge objekte ili funkcije. Objekti omogućavaju da grupišemo povezane podatke i funkcionalnosti. U JavaScript-u se često koristi pojam *JSON objekat* (JavaScript Object Notation) za literale objekata u kodu.

Kreiranje objekta literalom: Koriste se vitičaste zagrade `{ ... }`. Npr:

```
let osoba = {
  ime: "Petar",
  prezime: "Petrović",
  starost: 30,
  pozdrav: function() {
    console.log("Ćao! Ja sam " + this.ime);
  }
};
```

Ovaj objekat *osoba* ima svojstva: *ime*, *prezime*, *starost* i *pozdrav*. Primećujemo da **vrednost svojstva može biti i funkcija** – u tom slučaju svojstvo se zove *metod*. Ovde je *pozdrav* metod koji ispisuje poruku. Unutar metoda, `this` referiše na **trenutni objekat** (osoba na koju se metod odnosi). Tako `this.ime` unutar *pozdrav* metode uzima vrednost svojstva *ime* ovog objekta (*"Petar"*).

Pristup svojstvima: Dva načina – tačka notacija i uglaste zagrade. Tačka: `console.log(osoba.ime);` ispisuje "Petar". Uglaste: `console.log(osoba["prezime"]);` ispisuje "Petrović". Tačka je praktičnija i češća, dok se uglaste koriste kada ime svojstva imamo dinamički (npr. u promenljivoj) ili ako svojstvo ima razmake ili rezervisane reči (što se generalno izbegava u imenovanju). Možemo dodavati nova svojstva u objekt prosto dodeljivanjem: `osoba.adresa = "Beograd";`, ili menjati postojeća: `osoba.starost = 31;`. Brisanje svojstva: `delete osoba.prezime;`.

Objekti mogu **sadržati druge objekte** kao vrednosti svojstava, čime gradimo hijerarhije podataka (npr. objekat *firma* može imati objekat *adresa* sa poljima *ulica*, *grad*, itd.). Takođe, moguće je iterirati kroz sva svojstva objekta pomoću `for...in` petlje:

```
for (let kljuc in osoba) {
  console.log(kljuc + ": " + osoba[kljuc]);
}
```

Ovo će proći kroz sve *enumerabilne* osobine objekta *osoba* i ispisati ih (redosled nije garantovan, ali u praksi često odgovara redosledu definicije).

Prototype i nasleđivanje: JavaScript objekti imaju mehanizam prototipskog nasleđivanja. Svaki objekat ima internu vezu ka svom *prototipu*, od koga može nasleđivati svojstva. Ovo je naprednija tema, ali korisno je znati da mnogi objekti dele zajedničke prototipe: npr. svi nizovi dele prototip `Array.prototype` (što objašnjava zašto svaki niz „nasleđuje“ metode poput `push`, `map`, itd.), dok svi obični objekti potiču od `Object.prototype`. Možete pristupiti prototipu objekta preko `Object.getPrototypeOf(obj)` ili **proto** (iako se ova druga sintaksa više ne preporučuje). Nasleđivanje omogućava reuse koda – definisanjem metoda u prototipu, oni su dostupni svim instancama.

Klase (ES6): Da bi se olakšalo pravljenje više sličnih objekata, uveden je sintaksni šećer za *klase*. Klasu možemo shvatiti kao *šablon* za objekte sa određenim svojstvima i metodama. Primer klase:

```
class Korisnik {
  constructor(ime, email) {
    this.ime = ime;
    this.email = email;
  }
  pozdrav() {
    console.log(`Zdravo, ja sam ${this.ime}.`);
  }
}
let k = new Korisnik("Mika", "mika@example.com");
k.pozdrav(); // "Zdravo, ja sam Mika."
```

Ispod haube, klase u JS zapravo koriste prototipsko nasleđivanje (metodi definisani u class sintaksi zapravo idu u prototip). **Klase** čine kod čitljivijim kada radimo sa konstruktorima i objektima istog tipa. U početku učenja možete i bez klasa, fokusirajući se na objektne literale i razumevanje `this`. Kako napredujete, upoznaćete se detaljnije sa klasama, *constructor* funkcijama i prototipovima.

Rad sa DOM-om (Document Object Model)

Jedna od **najvažnijih primena JavaScript-a** je manipulacija **DOM**-a u web stranici. **DOM (Document Object Model)** je *programski interfejs za web dokumente* koji predstavlja HTML stranicu kao hijerarhiju objekata u vidu stabla [22]. Pregledač pri učitavanju stranice HTML kod prevodi u DOM stablo u memoriji – gde svaki HTML element postaje **čvor (node)** tog stabla (npr. `<body>` je čvor koji sadrži svoje child čvorove `<h1>`, `<p>`, itd.) [23] [24]. JavaScript kroz DOM može da pristupi svakom od ovih elemenata kao objektu i da **menja strukturu, sadržaj i stil stranice**. U suštini, DOM povezuje web stranicu sa JavaScript-om, omogućavajući skriptu da *dinamički ažurira* prikazani dokument.

Manipulacija DOM-a znači da JavaScript može:

- **Pronaći** bilo koji HTML element na stranici (preko ID, klase, taga, selektora...),
- **Izmeniti sadržaj** tog elementa (tekstualni ili HTML sadržaj),
- **Promeniti stil (CSS)** elementa (npr. sakriti element, promeniti mu boju, poziciju, klasu...),
- **Dodati ili ukloniti** elemente (dinamički kreirati nove čvorove, ili obrisati postojeće),
- **Reagovati na događaje** na elementima (klik, kretanje miša, unos teksta itd.),
- **Kreirati nove događaje** i pokrenuti ih programatski,

drugim rečima, sve što je potrebno za interaktivnu stranicu [25].

Da bismo radili sa DOM-om, prvo moramo **pristupiti elementima** koji nas interesuju. Najčešći načini:

- `document.getElementById("neki-id")` – vraća element sa datim **ID** atributom. Npr. ako HTML ima `<p id="opis">...</p>`, možemo u JS: `let paragraf = document.getElementById("opis");`.
- `document.querySelector("selektor")` – vraća **prvi element** koji odgovara CSS selektoru. Primer: `document.querySelector("div.item span")` – prva `<span>` unutar elementa sa klasom "item". `querySelectorAll` slično vraća *NodeList* svih elemenata koji odgovaraju selektoru.
- `document.getElementsByClassName("klasa")`, `getElementsByTagName("tag")` – vraćaju kolekcije elemenata po klasi ili imenu taga (stariji način, danas se često koristi querySelectorAll koji pokriva i ovo).

Kada imamo referencu na DOM element, možemo koristiti njegova svojstva i metode za manipuaciju:

Promena sadržaja: Svaki element ima svojstvo `innerHTML` koje predstavlja HTML sadržaj unutar tog elementa (kao string). Dodeljivanjem `innerHTML` možemo menjati ceo unutrašnji HTML. Primer: `paragraf.innerHTML = "<b>Novi sadržaj</b>";`. Time ćemo u paragraf ubaciti boldirani tekst. Ako želimo menjati **samo tekst** bez uticaja na eventualne child elemente, bolje je koristiti `textContent` ili `innerText`. `element.textContent = "plain text"` će zameniti tekstualni sadržaj elementa, ne tumačeći potencijalne HTML tagove (tretira ih kao običan tekst). Razlika `innerText` vs `textContent`: `innerText` obraća pažnju na vidljivost (neće uključiti skrivene elemente) i preračunava stil, dok `textContent` vraća *sve* tekstualne node-ove. U praksi za većinu slučajeva možete koristiti `textContent` za čitanje ili promenu tekstualnog sadržaja.

Promena atributa: Možemo pristupiti atributima kao svojstvima objekta – npr. `img.src = "neka_slika.png";`, `link.href = "https://drugi.url"` i sl. Ovo automatski menja i na stranici (npr. promeni sliku koja se prikazuje). Postoji i opšti metod `element.setAttribute(name, value)` i prateći `getAttribute`, ali za standardne atribute direktno svojstva uglavnom rade.

Promena stila: Svaki element ima objekat `style` koji predstavlja *inline* stilove tog elementa. Možemo menjati CSS kroz njega, npr: `element.style.backgroundColor = "yellow";`, `element.style.display = "none";` (sakriva element), itd. Ova svojstva prate tzv. *camelCase* notaciju za imena koja u CSS-u sadrže crticu (background-color -> backgroundColor). Ipak, često je praktičnije dodavati/uklanjati klase na elementu umesto podešavanja pojedinačnih stilova, jer dizajn obično držimo u CSS klasama. Možemo koristiti `element.classList.add("active")` ili `remove` za manipulaciju klasama.

Dodavanje novih elemenata: Za kreiranje novog elementa koristimo npr. `document.createElement("tagName")`. Ovo kreira novi DOM element zadatog tipa ali ga još ne postavlja u stablo. Potom možemo, recimo, kreirati i tekst čvor: `document.createTextNode("Neki tekst")`, ili umesto toga direktno koristiti `element.innerText = "tekst"` nakon kreiranja elementa. Kada pripremimo novi čvor, **dodajemo ga u DOM** preko metoda poput `parent.appendChild(noviElement)` ili `parent.insertBefore(noviElement, refElement)` za umetanje na određenu poziciju. Primer:

```
let lista = document.getElementById("listaKomentara");
let novi = document.createElement("li");
novi.textContent = "Novi komentar";
lista.appendChild(novi);
```

Ovaj kod bi dodao novi `<li>` na kraj liste sa ID "listaKomentara". Slično postoji i metoda `parentElement.removeChild(child)` ili za direktno uklanjanje sebe `element.remove()` (novija metoda) kako bismo obrisali čvor iz DOM-a.

Navigacija kroz DOM: Od datog elementa, možemo pristupiti njegovom roditelju (`elem.parentElement`), njegovoj deci (`elem.children` ili firstElementChild/lastElementChild), sledećem ili prethodnom sibling elementu (`elem.nextElementSibling`), itd. Postoji čitav API za kretanje kroz DOM stablo (`childNodes`, `firstChild` uključujući i tekstualne čvorove, itd.), ali u praksi se češće elementi dohvaćaju direktno selektorima nego ručnom navigacijom.

Ukratko, **DOM API** nam daje sve što je potrebno da JavaScript kod **dinamički menja web stranicu u runtime-u**. Kada kažemo da JS "oživi" stranicu – to se dešava upravo kroz DOM: menjanjem elemenata i reagovanjem na događaje.

> **Napomena:** DOM nije deo samog jezika JavaScript (nije definisan ECMAScript specifikacijom) već je posebna specifikacija (W3C DOM standard). Pregledači implementiraju DOM interfejse koje JavaScript kod može koristiti [26] [27]. Tako, kada kod napiše `document.querySelector(...)`, `document` objekat i njegove metode su deo browser API-ja (Web API), a ne ugrađenog JS jezika. Ovo praktično nije bitno za nas dokle god radimo unutar browsera – sve nam je transparentno dostupno – ali vredi znati da ako pokrenete JS u drugom okruženju (npr. Node.js), *document* i DOM neće postojati osim ako se ne emulira.

Događaji (Events) i upravljanje događajima

Događaji su ključni koncept interakcije – predstavljaju "dešavanja" na koja možemo reagovati kodom. Događaji mogu biti korisničke akcije (klik miša, pritisak tastera, pomeraj, scroll, slanje forme), ali i sistemski događaji (učitavanje stranice, završetak učitavanja slike, tajmer i sl.). U DOM-u, skoro svaki element može emitovati određene događaje.

Da bismo reagovali na neki događaj, potrebno je da definišemo **event handler** ili *listener* – funkciju koja će se pozvati kada do događaja dođe. Postoje tri načina u JS kako pridružiti handler događaju: **inline** (u HTML atributu, što se *ne preporučuje*), preko svojstva objekta (npr. `element.onclick = function() {...}`), ili moderni način `addEventListener` metoda:

```
const btn = document.querySelector("button#send");
btn.addEventListener("click", function(event) {
  alert("Kliknuto na send!");
});
```

U ovom primeru, dohvatili smo `<button id="send">` i dodali slušača na događaj "click". Funkcija će se izvršiti svakim klikom na dugme. Primetite da handler funkcija prima jedan argument `event` (koji ovde nismo eksplicitno koristili) – to je **Event objekat** koji nosi informacije o događaju (tip, cilj, koordinate klika i dr.). Možemo ga iskoristiti ako treba (npr. `event.target` referiše na element na kom se desio događaj, što je ovde zapravo *btn*).

Najčešći događaji i njihova primena: - `click` – klik lijevog tastera miša na element (za `button`, `a` tagove itd.).
- `dblclick` – dvostruki klik.
- `mouseover` / `mouseout` – prelazak miša preko elementa i izlazak.
- `mousedown` / `mouseup` – pritisak i otpuštanje tastera miša.
- `keydown` / `keyup` / `keypress` – događaji tastature za pritisak/otpuštanje tastera.
- `input` – svaki put kad se promeni vrednost input polja (korisnik unosi tekst).
- `change` – kada se element forme promeni i izgubi fokus (za checkbox odmah pri kliku).
- `submit` – pri slanju forme (može se iskoristiti `event.preventDefault()` unutar handlera da se spreči podrazumevano slanje i npr. izvrši AJAX poziv umesto reload stranice).
- `load` – kada se učita stranica ili neki resurs (npr. `<img>` ima load događaj kad je slika učitana).
- `DOMContentLoaded` – specijalan događaj na `document` objektu koji signalizira da je HTML parsiran i DOM stabla kreirano (koristan za izvršiti JS tek nakon što DOM postoji).

Postoji čitav spisak događaja za različite tipove objekata (prozor, dokument, elementi, XMLHttpRequest, itd.).

Uklanjanje event listenera: Ako smo dodali slušalac preko `addEventListener`, možemo ga kasnije ukloniti `removeEventListener` pozivom sa istim argumentima (moramo imati referencu na originalnu funkciju).

`this` **u handlerima:** Ako koristite `element.onclick = function(){ ... }`, unutar te funkcije `this` će referisati na *element*. Međutim, ako koristite arrow funkciju, ona nema svoj `this` pa će on biti iz spoljnog konteksta – to ponekad nije željeno ponašanje za event handler (zato se najčešće koriste

klasične anonimne funkcije ili odvojene imenovane funkcije za handlere, a arrow ređe osim u slučajevima kad vam `this` ne treba).

Event bubbling i propagation: Događaji u DOM hijerarhiji *mehuriću* na gore – npr. klik na `<button>` prvo pokreće handler na tom buttonu (ako postoji), zatim se taj isti događaj propagira na njegovog roditelja (npr. `<form>`), pa dalje na sve pretke do `document` objekta. Ovo omogućava *delegiranje događaja* – možete uhvatiti događaj na višem nivou umesto na samom elementu. Ako želite da sprečite propagaciju (npr. klik se ne prosleđuje parent elementu), u handleru pozovete `event.stopPropagation()`.

Asinhronost događaja: Važno je napomenuti da su event handleri suštinski asinhroni – oni se izvršavaju kao odgovor na spoljašnje dešavanje, ne tokom sekvencijalnog izvršavanja koda. Pregledač ima **event loop** mehanizam koji prati sve događaje i kada se neki desi, stavlja poziv handlera u red izvršenja. O ovome više u sledećem poglavlju o asinhronom JS-u. Bitno je razumeti da vaš glavni program ne "čeka" događaj; on se odvija, i kada se dogodi event, *prekid* se desi i izvrši se handler, pa se program nastavlja.

Dakle, putem događaja na stranici možemo interaktivno reagovati na korisnike – klik na dugme pokreće slanje podatka, pritisak tastera ažurira nešto u interfejsu u realnom vremenu, itd. Ovo je centralni koncept web programiranja sa JS.

Asinhroni JavaScript

Do sada smo pretpostavili *sekvencijalno* izvršavanje koda (linija po linija). Međutim, web aplikacije često moraju obavljati **dugotrajne ili odložene zadatke** (npr. čekati odgovor sa servera, sačekati 2 sekunde pa izvršiti akciju, ili reagovati na događaj koji će se desiti kasnije). Da bismo to ostvarili **bez blokiranja** ostatka programa, JavaScript koristi asinhroni model izvršavanja.

JavaScript runtime je jednonitni (single-threaded) – što znači da u datom trenutku može raditi samo jednu stvar [28]. Ako bismo pokušali da izvršimo veoma dugotrajnu operaciju sinhrono (npr. kompleksni proračun od par sekundi), tokom tog vremena *sve ostalo staje* – brauzer ne može da osveži interfejs, ne reaguje na klikove, itd. Zato se dugotrajni poslovi i spoljašnji zahtevi (poput mrežnih poziva) izvode **asinhrono** – tako da se započnu u pozadini, a glavna nit može da nastavi sa drugim radom (npr. interakcija sa korisnikom) [28].

U klasičnom smislu, asinhrono programiranje u JS se zasniva na **callback** funkcijama: umesto da funkcija vrati rezultat odmah, **prosledimo funkciju povratnog poziva** koja će se izvršiti kada rezultat bude spreman. Primer:

```
console.log("Početak");
setTimeout(function() {
  console.log("Ova poruka se pojavljuje posle 2 sekunde");
}, 2000);
console.log("Kraj");
```

Korišćenjem ugrađene funkcije `setTimeout`, prosledili smo callback koji ispisuje poruku. Kod izvan timeout-a se ne blokira – redom će se ispisati "Početak", **zatim odmah** "Kraj", a tek nakon ~2 sekunde ispisuje se poruka iz timeout callbacka. Ovaj mehanizam pokazuje kako JavaScript može *čekati* na nešto u pozadini, a da pritom ne zaustavlja interakciju ili ostale skripte. Funkcija `setTimeout` je primer

asinhrone API funkcije – ona raspoređuje zadatak (ovde timer) i odmah vraća kontrolu programskom toku, dok će prosleđenu funkciju pozvati kasnije preko event loop-a. Slično, **AJAX pozivi** (koristeći `fetch` API ili stariji `XMLHttpRequest`) funkcionišu asinhrono: pošaljemo zahtev ka serveru i definišemo callback ili *Promise* za odgovor, a ostatak koda teče dalje.

Event loop: Interno, JavaScript okruženje ima tzv. *event loop* (petlju događaja) koji nadgleda **queue (red) zadataka**. Kada se dogodi asinhroni događaj (npr. stigne odgovor sa servera, istekne timer, korisnik klikne nešto), tada se odgovarajući callback stavi u red. Event loop stalno proverava da li je glavni thread slobodan, i onda izvlači prvi zadatak iz reda i izvršava ga [29]. Tako se ostvaruje iluzija paralelizma – zaista se istovremeno ne izvršava više JS koda, već se komadi posla ređaju i izvršavaju jedan po jedan brzo čim uslovi dozvole.

Moderni pristupi asinhronosti: Callback funkcije su osnovni mehanizam, ali mogu dovesti do *"callback pakla"* kada se previše ugnjezde (teško za održavanje). ES6 je uveo **Promise** objekat – reprezentuje buduću vrednost ili grešku asinhrone operacije. Promise ima metode `.then` (za obradu rezultata) i `.catch` (za greške), čime omogućava lakše ulančavanje asinhronih operacija bez gnezda. Primer s `fetch` API:

```
fetch("podaci.json")
  .then(response => response.json())
  .then(data => {
    console.log("Stigli podaci:", data);
  })
  .catch(error => {
    console.error("Greška pri dohvatanju", error);
  });
```

Ovde `fetch` vraća Promise koji ide u pending state dok traje mrežni poziv. Kada (ako) stigne odgovor, prelazi u resolved state i poziva se prvi `.then` sa odgovorom. U njemu pretvaramo odgovor u JSON (što je opet asinhrono, vraća promise), pa sledeći `.then` dobija konačne podatke i može ih obraditi. Ako se bilo gde desi greška, `.catch` će je uhvatiti. Ovaj lanac je čitljiviji nego zagnježđeni callbackovi.

ES2017 (ES8) je doneo još jednu sintaksnu pogodnost: **async/await**. Ključna reč `async` ispred definicije funkcije označava da ta funkcija vraća **Promise**, a unutar nje možemo koristiti `await` ispred poziva promise-vraćajućih funkcija umesto `.then` lančanja. Kod iznad pisan sa async/await:

```
async function ucitajPodatke() {
  try {
    let response = await fetch("podaci.json");
    let data = await response.json();
    console.log("Stigli podaci:", data);
  } catch (error) {
    console.error("Greška:", error);
  }
}
```

`await` zaustavlja izvršavanje te async funkcije dok se Promise ne reši, ali **ne blokira** ostatak programa (samo tu async funkciju). Ovo omogućava da pišemo asinhroni kod koji izgleda slično sekvencijalnom,

što značajno olakšava razumevanje. Bitno je napomenuti: `await` se sme koristiti samo unutar `async function`.

Još primena asinhronosti: Event listener-i su svojevrsan asinhroni mehanizam – kao što smo gore objasnili, definišete funkciju koja će se zvati kad događaj nastupi, ali u međuvremenu program teče dalje. Takođe, JavaScript ima i `setInterval` (ponavljajući timer), WebSockets za real-time komunikaciju, rad sa datotekama i bazama u browseru (IndexedDB) – sve su to asinhrone operacije. Na serveru (Node.js) asinhronost je još češća (čitanje fajlova, umrežavanje itd. su asinhroni).

Zašto je ovo važno? Kao budući profesionalni programer, vrlo brzo ćete nailaziti na situacije gde morate čekati na rezultat (od servera najčešće) a da korisnički interfejs ostane responzivan. Razumevanje kako JavaScript upravlja asinhronim kodom i kako da pravilno koristite Promise i async/await je ključno. Zapamtite, **JavaScript izvršava sve u jednoj niti**, ali omogućava *konkurentnost* putem event loop-a i asinhronih callbackova – to je moćna osobina ali zahteva promišljen pristup.

(Za dublje razumevanje, možete kasnije istražiti detalje o Event Loop mehanizmu – npr. microtask queue vs macrotask queue, i kako JS rukuje obećanjima, ali za start je dovoljno shvatiti osnovni koncept iznad.)

Zaključak i naredni koraci

Ovaj **JS Starter Pack** pokrio je fundamentalne koncepte JavaScript-a sa kojima treba da budeš potpuno upoznat uvek – promenljive, tipove i kako funkcioniše dinamičko tipiziranje, osnovne operatore, kontrolu toka sa uslovima i petljama, deklarisanje i korišćenje funkcija (uključujući modernu sintaksu), rad sa nizovima i objektima, kao i osnove interakcije sa web stranicom preko DOM-a i događaja. Takođe smo uveli i pojam asinhronog programiranja, koji postaje neizbežan u realnim primenama (komunikacija sa serverom, tajmeri, UI događaji i sl.).

Kao što si napomenuo, cilj je **realna primena** – fokusirali smo se na savremene konstrukte (npr. `let/const`, arrow funkcije, `querySelector`, fetch API sa promise-ima) i zaobilazili zastarele obrasce (poput `var`, `alert` za sve osim najjednostavnijih testova, stari `XMLHttpRequest`, itd.) koje u modernom razvoju retko treba koristити. Naravno, sve u JS ima svoje mesto, ali ograničeno vreme bolje je utrošiti na ono što će se zaista koristiti svakodnevno. **U profesionalnom okruženju**, očekuje se da dobro barataš ovim osnovama jer će se one konstantno pojavljivati – bilo da pišeš čisti JS kod ili koristiš front-end frejmvorke (koji su opet izgrađeni na ovim osnovama).

Sledeći koraci za učenje/praktikovanje: Preporučljivo je da sada, nakon teorije, kreneš da vežbaš svaku od ovih tema na malim primerima. Na primer, probaj da napišeš funkcije za računanje nečega, manipuliši nizovima (dodaj/ukloni element, filtriraj, sortiraj), kreiraj objekat koji ima metode, napravi malu HTML stranicu gde ćeš pomoću JS dodavati elemente, menjati stil, i reagovati na neko dugme klikom (koristeći `addEventListener`). Kroz takvu praksu konsolidovaćeš teoriju. Uvek se vraćaj ovom dokumentu da se podsetiš sintakse ili koncepta koji ti nije jasan – napravili smo sekcije tako da lako pronađeš ono što ti treba.

U *sledećoj fazi*, možemo generisati niz vežbi i zadataka koji će ti pomoći da primeniš ova znanja u praksi, jer programiranje se najbrže uči radeći. Posebno ćemo se fokusirati na *real-world* primere, poput manipulacije DOM-a (jer to je ono što web razvoj čini vidljivim i interaktivnim), rad sa vremenskim funkcijama, kao i simulaciju jednostavne komunikacije sa serverom (npr. dohvat podataka sa neke public API). Takođe, nastavićemo da nadograđujemo znanje – teme poput **obrada grešaka (try/catch)**, **lokalno skladištenje (LocalStorage)**, modularizacija koda (ES6 moduli), pa i uvod u neki frejmvork (React, Angular ili sl.) biće lakše savladati kad čvrsto stoje osnove koje smo ovde pokrili.

Rezime: JavaScript je moćan upravo zato što omogućava sve ovo raznoliko ponašanje – od manipulacije web stranicom u brauzeru do kompleksne logike aplikacije – a sada imaš solidnu teorijsku osnovu. Drži ovaj dokument pri ruci (_bookmark_uj ga u svojoj glavi za "svako doba dana i noći" kako si tražio) i nadograđuj ga beleškama kroz praksu. Sledeće krećemo sa praktičnim vežbama koje će te uvesti u rutinirano korišćenje ovih koncepata. Srećno kodiranje!

Literatura / Reference:

- *MDN Web Docs – JavaScript Guide & Reference – (engleski)* zvanična Mozilla dokumentacija sa detaljnim opisima JS jezika i DOM API-ja. Preporuka za dalje čitanje: poglavlja o osnovama sintakse [1], tipovima podataka [10] [11], konverzijama [12] [13], kao i DOM uvod [22].
- *freeCodeCamp – Var, Let, and Const – What's the Difference?* – članak koji sumira razlike između var/let/const [2] (ovde smo saželi ključne tačke).
- *W3Schools – JavaScript HTML DOM* – pregled praktičnih mogućnosti DOM manipulacije (dodavanje/menjanje elemenata, itd.) [25].
- *Škola koda (blog)* – članci na našem jeziku, npr. o streličastim funkcijama [30].
- *Teknasyon Engineering blog* – "What is the DOM & How does HTML Rendering happen" – objašnjenje DOM koncepta i prikaz DOM stabla grafički (korisno za razumevanje strukture) [23] [24].
- *Eloquent JavaScript (Marijn Haverbeke)* – besplatna online knjiga (i na srpskom prevod postoji) koja dubinski obrađuje JS; poglavlja 1-5 pokrivaju osnove slične ovde opisanim, a dalje ide u naprednije teme.

---

[1] [10] [11] [12] [13] [15] [16] [17] Grammar and types - JavaScript | MDN
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_types

[2] [3] [4] [5] [6] [7] [8] [9] Var, Let, and Const – What's the Difference?
https://www.freecodecamp.org/news/var-let-and-const-whats-the-difference/

[14] Expressions and operators - JavaScript - MDN Web Docs
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_operators

[18] Equality comparisons and sameness - JavaScript | MDN
https://lia.disi.unibo.it/materiale/JS/developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_sameness.html

[19] [20] [21] [30] Škola koda | Streličaste (arrow) funkcije
https://skolakoda.github.io/strelicaste-funkcije

[22] [23] [24] [26] [27] Document Object Model (DOM) - Web APIs | MDN
https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model

[25] JavaScript HTML DOM
https://www.w3schools.com/js/js_htmldom.asp

[28] [29] Introducing asynchronous JavaScript - Learn web development | MDN
https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Async_JS/Introducing