ChatGPT

Mentorski vodič: JavaScript – Dan 1 i Dan 2

Uvod: Ovaj vodič obuhvata prve dve lekcije *"JavaScript Roadmap – 30 dana do Junior Full-Stack Developera"*. Kroz Dan 1 i Dan 2 proći ćemo osnove postavke radnog okruženja i uvođenja u JavaScript, na način koji simulira realno radno okruženje. Radićemo praktične zadatke u Visual Studio Code okruženju (sa instaliranim ESLint-om i drugim korisnim ekstenzijama), umesto pasivnog gledanja videa. Cilj je da kroz pisanje koda i objašnjenja "ispod haube" steknete početno iskustvo kao junior programer. Svi koraci, fajlovi i vežbe predstavljeni su kao na pravom poslu, uz dobre navike programiranja.

Dan 1: Postavka okruženja i „Hello World"

Cilj dana: Pripremiti razvojno okruženje i isprobati pokretanje prve JavaScript skripte. Naučićete kako da ubacite JavaScript u web stranicu, kako da pokrenete kod u pregledaču i putem Node.js, te kako da koristite konzolu za debugovanje. Takođe ćemo objasniti razlike između izvršavanja skripti u browser-u i u terminalu (Node okruženje) [1] . Krenimo redom, kao da smo na prvom radnom zadatku.

Radno okruženje: VS Code, Node.js i ESLint

U realnom okruženju, programeri najčešće koriste moderni editor kao što je **Visual Studio Code (VS Code)**. Zato prvo postavljamo VS Code i pripremamo okruženje:

- **Instalacija alata:** Uverite se da su instalirani VS Code i **Node.js** (platforma za izvršavanje JS koda na serveru). Node.js ćemo koristiti i za razvojne alate, i za pokretanje JS skripti u terminalu radi poređenja sa browser-om.
- **Ekstenzije u VS Code-u:** Kao junior developer, dobićete pred-konfigurisano okruženje ili uputstvo da instalirate korisne ekstenzije. Preporučujemo da instalirate **ESLint** ekstenziju (za hvatanje grešaka u kodu i poštovanje stilskih pravila) [2] . Takođe, ekstenzije poput **Live Server** (za pokretanje lokalne stranice) i **Prettier** (za automatsko formatiranje koda) mogu biti od pomoći.
- **Projekat i fajlovi:** Napravite novi direktorijum (projekat) za vežbu, npr. `js-day1`. U VS Code-u otvorite taj folder. Kreirajte prazan fajl `index.html` i prazan fajl `script.js`. Ove datoteke će imitirati tipičnu strukturu projekta gde je HTML za strukturu stranice, a JS u posebnom fajlu za logiku.

Uključivanje JavaScript-a u HTML (inline vs. eksterno)

Postoje dva načina da dodamo JavaScript u HTML stranicu: **inline** (direktno unutar `<script>` taga u HTML fajlu) ili **eksterno** (kao poseban `.js` fajl koji se povezuje u HTML-u). U profesionalnom okruženju preferira se eksterni fajl, jer odvajanje koda olakšava održavanje.

- **Inline skripta:** U HTML fajlu možete napisati `<script> console.log("Hello World"); </script>` direktno, i ta skripta će se izvršiti pri učitavanju stranice. Ovo je korisno za manje testove, ali nije pregledno za veći kod.
- **Eksterni fajl:** Češće se koristi poseban JS fajl. U HTML-u se onda uputi na njega preko `<script src="script.js"></script>`. Ovakav pristup odvaja prezentaciju (HTML) od logike (JS), što je dobra praksa.

Primer minimalne HTML stranice koja uključuje eksterni JavaScript fajl:

```
<!-- index.html -->
<!DOCTYPE html>
<html lang="sr">
<head>
  <meta charset="UTF-8" />
  <title>JavaScript Dan 1</title>
</head>
<body>
  <h1>Moja prva JS stranica</h1>

  <!-- Povezivanje spoljašnjeg JS fajla -->
  <script src="script.js"></script>
</body>
</html>
```

U ovom primeru, unutar `<body>` na dnu dodajemo `<script src="script.js"></script>`. To znači da će se kod iz `script.js` fajla učitati i izvršiti kada browser dođe do tog taga. (Napomena: Skriptu obično stavljamo pred kraj `body` sekcije kako bi se HTML elementi iznad nje već učitali u DOM pre nego što skripta pokuša da manipuliše njima.)

Inline vs eksterno: Inline skripte mogu biti zgodne za vrlo jednostavne stvari ili za demonstracije, ali u realnosti želećete gotovo sav kod u izdvojenim `.js` fajlovima. To omogućava višekratnu upotrebu koda, bolju organizaciju i primenu alata poput lintera.

Pokretanje lokalne stranice u pregledaču

Nakon što smo kreirali HTML i povezali JS, vreme je da otvorimo stranicu i vidimo da li radi. Postoji više načina da pokrenemo HTML stranicu lokalno:

- **Otvaranje HTML fajla direktno:** Najjednostavnije, možete dvokliknuti na `index.html` fajl ili ga otvoriti u pregledaču prevlačenjem u prozor pregledača. Pregledač će prikazati stranicu. Međutim, ponekad zbog bezbednosnih ograničenja neki JS funkcionalnosti ne rade kada je fajl otvoren sa `file://` putanjom.
- **Korišćenje Live Server-a:** Bolja opcija u VS Code-u je da koristite ekstenziju **Live Server** (ako ste je instalirali). Desni klik na `index.html` i odaberite "Open with Live Server". Ovo će pokrenuti mali lokalni server i otvoriti stranicu na adresi poput `http://127.0.0.1:5500/index.html`. Prednost je što se stranica automatski osvežava kad god sačuvate izmene, i simulira uslove slične pravom web serveru.
- **Pokretanje kroz Node (nije za HTML direktno):** Node.js nije tipično sredstvo za “otvaranje” HTML stranice, jer Node je okruženje za izvršavanje JS koda van pregledača. Da biste videli stranicu u pregledaču, koristite metode iznad. Node.js ovde nećemo koristiti za posluživanje HTML-a (za to služe web serveri ili Live Server tokom razvoja).

Nakon pokretanja stranice u pregledaču, trebalo bi da vidite sadržaj stranice (naslov „Moja prva JS stranica“). Nećete videti odmah “Hello World” poruku, jer je ona planirana da se ispiše u konzoli kroz `console.log` (objasnićemo to uskoro). Da biste videli rezultate JS koda, trebaće vam **DevTools konzola**.

Pokretanje skripte kroz Node.js

Pored pregledača, JavaScript kod se može izvoditi i direktno u Node.js okruženju (terminal). Ovo je koristan eksperiment da vidite razliku između browser JS i Node JS [1]. Na primer, ako u fajlu `script.js` imate:

```
// script.js
console.log("Hello from Node!");
```

Možete otvoriti terminal, navigirati do foldera projekta i pokrenuti komandu:

```
node script.js
```

Ova komanda će izvršiti `script.js` koristeći Node.js i odštampaće rezultat u terminalu. Ako vidite u terminalu `Hello from Node!`, uspešno ste pokrenuli JS kod van pregledača.

Razlike browser vs Node: Kada isti `console.log("Hello World")` kod pokrećete u pregledaču (putem HTML stranice) i u Node-u, dobićete ispis na dva različita mesta: u pregledaču u DevTools konzoli, a u Node-u direktno u terminalu. Međutim, najveća razlika je u dostupnim objektima i API-jevima: u browseru postoji globalni objekat `window` i `document` (DOM) – omogućava manipulaciju stranicom, dok u Node.js ti objekti **ne postoje**. Node ima svoj globalni objekat (npr. `global`), module, `process` objekat itd. Dakle, **browser skripte** su za interakciju sa web stranicom i korisnikom, dok **Node skripte** služe za server-side logiku, skripting, rad sa fajlovima itd. (U ovoj ranoj fazi fokusiramo se na pokretanje koda u pregledaču [1], a Node koristimo samo za eksperimente i kasnije alate.)

Rad sa DevTools konzolom (console.log i osnovno debugovanje)

Web pregledači (Chrome, Firefox, Edge itd.) imaju ugrađene razvojne alate (**Developer Tools** ili skraćeno DevTools). Da biste im pristupili, otvorite stranicu u pregledaču i pritisnite **F12** (ili `Ctrl+Shift+I` / `Cmd+Option+I` na Mac). Otvoriće se prozor sa više tabova – najvažniji za nas sada je **Console** (Konzola).

console.log: U JavaScript kodu, funkcija `console.log()` služi da ispišemo informacije u konzolu (za debugovanje ili praćenje toka programa). U našem `script.js` možemo napisati:

```
// script.js
console.log("Hello World!");
```

Kada osvežimo stranicu (ili otvorimo kroz Live Server), DevTools konzola u pregledaču će prikazati poruku **Hello World!**. Ovo je vaš prvi "output" u JavaScript-u. Često se **Hello World** koristi kao probni program da proverimo da li sistem radi.

Sada, u konzoli možete i interaktivno upisivati JavaScript izraze. Probajte u Console tabu direktno ukucati npr. `2 + 2` i pritisnite Enter – trebalo bi da dobijete rezultat `4`. Konzola je izvršno okruženje: možemo testirati kratke komande bez potrebe da ih pišemo u kod. Ovo je odlično za **eksperimentisanje i debugovanje**.

Debugging (pronalaženje grešaka): Ako vaš kod ima grešku, npr. sintaksnu grešku (propustili ste zagradu, tačku-zarez i sl.) ili koristite promenljivu koja nije definisana, u konzoli će se pojaviti **crvena greška**. Pročitajte pažljivo tu poruku – ona ukazuje na tip greške i liniju fajla gde je nastala. Na primer, ako u `script.js` napišemo `console.log(Hello World!)` bez navodnika, konzola će prikazati grešku o neprepoznatom tokenu. Ispravite grešku (dodajte navodnike) i osvežite stranicu.

Takođe, DevTools omogućava da postavite *breakpoint* (tačku prekida) u kodu i korak po korak pratite izvršavanje, ali za početak je dovoljno da znate čitati poruke u konzoli i koristiti `console.log` za ispis vrednosti promenljivih tokom razvoja.

ESLint i analiza grešaka

U profesionalnom kodiranju, vrlo je bitno držati se određenih pravila stila i hvatanja potencijalnih problema pre pokretanja koda. Tu pomaže **ESLint** – alat za statičku analizu koda. Pretpostavimo da ste u VS Code instalirali ESLint ekstenziju i u projektu imate fajl konfiguracije (npr. `.eslintrc`). To može biti već podešeno na projektu od strane mentora ili tima.

Kako ESLint radi: Dok kucate kod, ESLint proverava vaš kod u skladu sa pravilima definisanim u `.eslintrc`. Ako detektuje prekršaj pravila ili potencijalnu grešku, podvlači kod crvenom ili žutom talasastom linijom i često daje poruku o kakvom se problemu radi. Ovo vam omogućava da ispravite grešku pre nego što pokrenete kod.

- *Primer 1:* Zaboravili ste tačku-zarez ili zarez u nizu – ESLint će to označiti (u zavisnosti od pravila, npr. AirBnB stil zahteva tačku-zarez na kraju svake izjave).
- *Primer 2:* Deklarisali ste promenljivu koju nigde ne koristite – ESLint može dati upozorenje da promenljiva nije iskorišćena (što može ukazivati na suvišan kod ili tipfeler u imenu druge promenljive).
- *Primer 3:* Pokušavate da koristite promenljivu koju niste deklarisali – ESLint će odmah prijaviti `'<ime_promenljive>' is not defined.` To je izuzetno korisno za hvatanje grešaka u kucanju (typo). Na primer:

```
let poruka = "Dobro jutro";
console.log(poruka);    // ispravno: koristi 'poruka'
console.log(porukaaa);  // ESLint će prijaviti grešku: 'porukaaa' nije
definisana
```

ESLint bi iznad `porukaaa` istakao grešku (pre pokretanja), a ako biste pokrenuli kod bez lintera, dobili biste **ReferenceError** u konzoli. Uočite kako vas linter spasava od pokretanja buga.

- *Primer 4:* Pravila za kod: ESLint može nametati dobra pravila kao što su zabrana korišćenja `var` (zastarelo – umesto toga `let` / `const`), obavezno korišćenje `===` umesto `==`, maksimalna dužina linije, itd. Poštujte ova upozorenja jer su tu da unaprede kvalitet koda.

  Savet: Redovno gledajte ESLint upozorenja dok kodirate. Ispravljajte ih odmah – tako ćete izbeći nagomilavanje problema. Ako niste sigurni zašto je nešto greška, pitajte mentora ili pretražite opis pravila online. Linter ne zna uvek kontekst, ali uglavnom signalizira stvarne probleme ili nedoslednosti.

Kreiranje prve HTML stranice i povezivanje `script.js`

Sada ćemo praktično primeniti sve gore navedeno i napraviti "Hello World" korak-po-korak, kao da vam je mentor dao prvi zadatak:

1. **Struktura projekta:** U VS Code, u otvorenom folderu `js-day1`, napravite datoteku `index.html` i folder `js` (za skripte). Unutar `js` foldera, napravite datoteku `script.js`. Ova organizacija (HTML u root, JS u potfolderu) je česta na poslu kako bi se odvojili različiti tipovi fajlova.
2. **HTML kostur:** U `index.html` upišite osnovni HTML5 kostur. VS Code olakšava – kucajte `!` i pritisnite Tab, dobićete predložak. Dodajte u `<body>` neki vidljiv element, npr. `<h1>JavaScript vežba</h1>`, da imate nešto na stranici. Zatim, odmah pre zatvarajućeg `</body>` taga, dodajte liniju `<script src="js/script.js"></script>` da povežete vaš JS fajl.
3. **Hello World skripta:** Otvorite `js/script.js` i tu dodajte jednostavnu JavaScript naredbu:

   ```
   console.log("Hello World!");
   ```

   Sačuvajte fajl. (Primetićete da VS Code zbog ESLint-a možda formatira navodnike ili dodaje tačku-zarez automatski ako imate podešeno – to je u redu.)
4. **Pokretanje u pregledaču:** Otvorite `index.html` u pregledaču. Preporučeno je preko Live Server-a: desni klik na `index.html` > "Open with Live Server". Stranica će se otvoriti na lokalnoj adresi (npr. `http://127.0.0.1:5500/index.html`). Videćete naslov "JavaScript vežba". Sada otvorite DevTools konzolu (F12) i trebalo bi da vidite ispis: **Hello World!**. Čestitamo – vaša prva skripta radi!
5. **Provera u Node okruženju:** Sada probajte i sledeće – otvorite terminal (VS Code Terminal ili komandna linija) u projektu i pokrenite komandu:

   ```
   node js/script.js
   ```

   Ovo će izvršiti isti fajl ali kroz Node.js. U terminalu bi trebalo da bude ispis **Hello World!**. To potvrđuje da osnovni JS kod može da se izvrši i van pregledača. Međutim, imajte na umu da ako biste u `script.js` dodali npr. `document.querySelector('h1')`, to bi radilo u browseru (jer postoji `document`), ali u Node-u biste dobili grešku (jer `document` ne postoji). Zato se i naglašava razlika okruženja (DOM vs Node).
6. **Eksperimenti u konzoli:** Za kraj dana 1, u otvorenoj konzoli u pregledaču probajte neke osnovne stvari:
7. Ukucajte `console.log("Proba")` direktno u konzoli i pritisnite Enter – ispisuje "Proba".
8. Ukucajte varijantu: `alert("Pozdrav!")` – trebalo bi da dobijete popup prozorčić u pregledaču sa porukom (ovo je ugrađena funkcija browser-a, još jedan način interakcije, ali `alert` se ređe koristi u realnom kodu jer blokira interfejs).
9. Probajte da definšete promenljivu u konzoli: npr. `let x = 42` Enter, zatim `x + 8` Enter – konzola će zapamtiti vašu promenljivu i dati rezultat 50.
10. Ako nešto ne funkcioniše, proverite konzolu za greške (crveno). Debugujte kao što smo opisali: ispravite kod, osvežite i ponovo testirajte.

Resursi za Dan 1: Za dodatno čitanje i vežbu, pogledajte MDN članak **"What is JavaScript"** (uvod u JavaScript i kako se koristi u stranici) [3]. Takođe, interaktivni kurs **Scrimba – Intro to JavaScript** može biti koristan za uvodne koncepte [4].

Dan 2: Promenljive i tipovi podataka

Cilj dana: U drugoj lekciji fokusiramo se na **promenljive** i **tipove podataka** u JavaScript-u. Naučićete kako se deklarišu promenljive (`let`, `const`, `var`), koje sve vrste vrednosti možemo čuvati u promenljivama (brojevi, stringovi, booleans itd.), kako da proverite tip vrednosti (`typeof` operator), te kako rade osnovni operatori nad tim vrednostima. Takođe ćemo isprobati konverzije tipova (npr. pretvaranje stringa u broj) i ukazati na posebne vrednosti poput `NaN` i `Infinity`. Sve ovo pratićemo praktičnim primerima i zadacima [5].

Deklaracija promenljivih: `var`, `let` i `const`

Promenljive su imenovane "kutije" u koje skladištimo podatke. U JavaScript-u imamo tri ključne reči za deklaraciju promenljivih:

- `var` – stariji način deklaracije promenljive (ES5 i ranije). Danas se `var` retko koristi jer ima nekoliko *zamki*: `var` nema blokovski opseg već funkcijski opseg (što znači da ignoriše npr. `{ }` blokove osim funkcija), može se ponovo deklarisati u istom opsegu bez greške, i podržava hoisting (podizanje deklaracije na vrh), što ponekad dovodi do konfuzije. Zbog ovih razloga, moderni JS standardi su uveli bolje načine.
- `let` – uveden u ES6 (2015). Koristi se za promenljive koje **mogu menjati vrednost**. Ima blokovski opseg (vredi samo unutar bloka `{ }` u kom je definisan), ne dozvoljava ponovno deklarisanje u istom opsegu (što sprečava nenamerna dupliranja imena), i ne možemo ga koristiti pre nego što ga definišemo (za razliku od `var` koji bude `undefined` ako se koristi pre deklaracije). U praksi, `let` je standard za promenljive koje ćete menjati.
- `const` – takođe uveden u ES6. Označava **konstantu**, odnosno promenljivu kojoj nakon inicijalizacije ne možemo promeniti referencu/vrednost. Mora biti inicijalizovana prilikom deklaracije (`const pi = 3.14;` je u redu, ali `const pi; pi = 3.14;` nije). `const` deli ostale karakteristike sa `let` (blokovski opseg, nema hoisting upotrebe pre deklaracije). U praksi, koristite `const` kad god znate da se vrednost neće menjati (što je često – npr. za konfiguracije, fiksne reference, rezultate selektora itd.), a `let` za promenljive koje će menjati vrednost. Tako dobijate i čitljiviji kod i izbegavate greške.

Najbolja praksa: U modernom JS kodu gotovo uopšte ne koristimo `var`. Svaki `var` može biti zamenjen sa `let` ili `const` uz odgovarajuću prilagodbu. Podrazumevano, koristite `const` – pa tek ako vidite da ćete morati promeniti vrednost, onda koristite `let`. Ovo čini kôd "konzervativnijim": ako je nešto `const`, odmah znamo da mu se vrednost ne menja i svaka kasnija promena te promenljive bi bila greška. ESLint obično ima pravilo da preferira `const` kad god je moguće.

Evo nekoliko primera koji ilustruju razlike između `var`, `let` i `const`:

```
// Primer opsega (scope)
if (true) {
  var staro = "Var promenljiva";
  let novo = "Let promenljiva";
}
console.log(staro); // "Var promenljiva" – var "procuri" iz bloka
console.log(novo);  // ReferenceError – let ne postoji van bloka

// Primer redeklaracije
```

```
var staro = "Ponovo
var";   // dozvoljeno, redeklaracija u istom opsegu (može izazvati zabunu)
let novo = "Nova vrednost"; //   Greška: već postoji promenljiva 'novo' u
ovom opsegu

// const primer
const PI = 3.14159;
PI = 3.14; //   TypeError: ne može se promeniti konstanta

// Ne možemo koristiti let/const pre deklaracije:
// console.log(x); //   ReferenceError: Cannot access 'x' before
initialization
let x = 5;
```

Kroz ove primere vidimo: - `var staro` definisan unutar if-bloka dostupan je i van bloka (što često nije željeno ponašanje). - `let novo` postoji samo unutar bloka. - Ponovno deklarisanje `var staro` je moguće (može pregaziti vrednost slučajno), dok `let novo` pri drugom definisanju baca grešku – što nas štiti. - `const PI` kada pokušamo promeniti, dobijemo grešku – što je ispravno jer pi treba ostati konstanta.

Hoisting napomena: I `var` i `let/const` imaju ponašanje hoistinga (interno se deklaracije pomeraju na vrh funkcije ili skripte), ali razlika je što kod `var` dobijamo vrednost `undefined` ako ga koristimo pre var linije, dok kod `let/const` dobijamo grešku (tzv. temporal dead zone). U praksi: **uvek definišite promenljive pre upotrebe**, to rešava sve probleme.

Tipovi podataka u JavaScript-u

JavaScript je jezik sa **dinamičkim tipiziranjem** i podržava nekoliko osnovnih (primitivnih) tipova podataka. “Dinamičko” znači da promenljiva može sadržati vrednost bilo kog tipa i ta vrednost (a samim tim i tip) se može promeniti tokom izvršavanja. Za razliku od statički tipiziranih jezika, ne morate unapred deklarisati tip promenljive – on se određuje na osnovu vrednosti koja je dodeljena.

Osnovni tipovi (primitivi): JavaScript ima **8 osnovnih tipova** od kojih je **7 primitivnih** i **1 ne-primitivni** (objekti). Primitivni tipovi su elementarne vrednosti, dok su objekti složenije strukture. Evo pregleda primitivnih tipova i čemu služe:

- **Number (Broj):** Predstavlja numeričku vrednost. U JavaScript-u *svi* brojevi (bilo cijeli ili realni sa decimalom) su jednog tipa `number` (64-bitni floating point, IEEE 754 standard). Primeri: `42`, `3.14`, `-7`. Postoje i neke posebne numeričke vrednosti: `Infinity` (beskonačnost, npr. rezultát 1/0), `-Infinity`, i `NaN` (“Not-a-Number”, npr. rezultat nemogućih operacija poput 0/0 ili parseInt("abc")).
  (Napomena: Od ES2020 postoji i odvojeni tip za celobrojne vrednosti van 64-bitnog opsega – `BigInt` – o njemu ćemo niže.)
- **String (String/Niska):** Predstavlja tekstualni podatak, niz karaktera. String literal se piše pod navodnicima: **jednostrukim** `'tekst'`, **duplim** `"tekst"` ili backtick navodnicima `` `tekst` `` (backtick omogućava template literale i višelinijski tekst). Primeri: `"Hello"`, `'Đorđe'`, `` `Ovo je ${1+1} string` ``. Stringovi su **immutable** (nepromenljivi) – svaka operacija koja menja string ustvari kreira novi string (primitivna vrednost se ne može izmeniti in-place).

- **Boolean (Logički tip):** Ima dve moguće vrednosti: `true` (tačno) i `false` (netačno). Koristi se za logičke uslove, poređenja i kontrolu toka (if uslovi, petlje). Primer: `let isLoading = false;` (flag da li se nešto učitava).
- **Undefined:** Označava "nedodeljenu" vrednost. Promenljiva koja je deklarisana, ali joj nije dodeljena vrednost, ima vrednost `undefined` podrazumevano. Takođe, ako funkcija nema `return` naredbu, implicitno vraća `undefined`. Primer:

  ```
  let a;
  console.log(a); // undefined (jer a nema vrednost)
  ```

  `undefined` je zapravo vrednost tipa "undefined". (Postoji i globalna promenljiva `undefined` koja ima tu vrednost – ali nemojte je menjati.)
- **Null:** Null je literalni tip koji predstavlja **nameran izostanak vrednosti** (prazninu). To je posebna vrednost koja se obično koristi da označi "nema vrednost" ili "prazno". Na primer: `let trenutnoNema = null;`. Iako izgleda slično undefined-u, `null` se uglavnom programerski eksplicitno dodeljuje da pokaže da nešto nema sadržaj.
  Važno: `null` je tip sam za sebe (primitivni tip), ali **zbog istorijske greške** `typeof null` vraća `"object"` [6]. Dakle, iako je `null` primitivna vrednost koja znači "ništa", JavaScript će je pogrešno okarakterisati kao objekat. To je poznata greška u jeziku koja ostaje zbog kompatibilnosti [6] (uvek imajte na umu da `null` nije objekat, uprkos tom rezultatu).
- **Symbol:** Symbol je posebna vrsta primitiva uvedena u ES6. Predstavlja jedinstven identifikator. Svaki Symbol je jedinstven i nejednak ijednom drugom Symbol-u, čak i ako imaju isti opis. Koristi se uglavnom za posebne ključeve u objektima koji neće kolidirati sa drugim ključevima, ili za označavanje interne identifikacije. U početku, simboli nisu toliko potrebni junioru, ali dobro je znati da postoje. Primer: `const id = Symbol("id");`.
- **BigInt:** BigInt je novi tip (ES2020) koji omogućava predstavljanje celih brojeva proizvoljne veličine (većih od maksimalnog safe integer opsega za Number, koji je 2^53 - 1). BigInt se zapisuje dodavanjem **n** na kraj literala broja, npr. `9007199254740993n`. Ovaj tip vam neće trebati dok ne radite sa izuzetno velikim celobrojnim vrednostima (kriptografija, big data, itd.), ali dobro je znati da postoji.

Osim ovih primitivnih, postoji i **tip Object** (objekat) koji obuhvata sve složene strukture: nizove (Array), funkcije (Function), obične objekte `{}`, itd. Objekti su **mutabilni** (promenljivi) i ponašaju se po referenci. Njih ćemo detaljnije pokriti kasnije, nakon što savladamo primitivne tipove. Za sada, znajmo da su svi tipovi osim objekata **vrednosni tipovi (value types)** i da se prenose po vrednosti.

Provera tipa – `typeof`: JavaScript ima operator `typeof` koji vraća string sa nazivom tipa date vrednosti. Primeri:

```
console.log(typeof 42);         // "number"
console.log(typeof "Hello");    // "string"
console.log(typeof true);       // "boolean"
console.log(typeof undefined); // "undefined"
console.log(typeof null);       // "object" (posebna greška u jeziku, null
NIJE objekat) 6
console.log(typeof Symbol());  // "symbol"
console.log(typeof 123n);       // "bigint"
```

Kao što vidimo, `typeof` je koristan da brzo proverimo sa kakvim tipom imamo posla. Imajte na umu: - Za većinu primitiva dobićemo očekivane nazive. - Za funkcije, `typeof` vraća `"function"` (funkcije su objekti ali specijalno ih razlikuje). - Za nizove i objekte, `typeof` vraća `"object"` (pa kako onda da razlikujemo niz od objekta? To ćemo kasnije – postoji `Array.isArray()` za nizove, dok `typeof` ne razlikuje). - Za `null` dobijamo `"object"` zbog pomenute greške – pa tip null vrednosti proveravamo obično eksplicitno (`vrednost === null`).

Dinamičko tipiziranje – primer: Pošto je JavaScript dinamički tipiziran, ista promenljiva može tokom izvršavanja držati različite tipove vrednosti:

```
let podatak = 5;
console.log(typeof podatak); // "number"
podatak = "pet";
console.log(typeof podatak); // "string"
podatak = true;
console.log(typeof podatak); // "boolean"
```

Ovo je istovremeno fleksibilnost ali i potencijalni izvor grešaka ako se ne pazi. Dobra praksa je **ne menjati** tip koji promenljiva drži bez potrebe (npr. nemojte koristiti jednu promenljivu za potpuno različite stvari u različitim kontekstima).

Konverzije tipova (prevodjenje vrednosti)

Ponekad ćemo morati da konvertujemo vrednost iz jednog tipa u drugi – recimo, string koji sadrži broj u pravi broj tipa Number, ili broj u string radi prikaza. U JavaScript-u imamo **implicitne** (automatske) i **eksplicitne** (namerno izazvane) konverzije.

- **Implicitne konverzije:** Dešavaju se kada JavaScript očekuje neki tip, pa pokuša da konvertuje vrednost. Npr. u sabiranju stringa i broja, broj će se **implicite** pretvoriti u string. Ili u poređenju ne-strogim (`==`), dešavaju se razne konverzije (koje mogu biti izvor bugova, zato koristimo `===`). Trudićemo se da **izbegavamo implicitne konverzije** jer mogu biti nejasne. Umesto toga, radije jasno konvertujte vrednost pre operacije – to čini kod razumljivijim.
- **Eksplicitne konverzije:** Koristimo funkcije/konstruktore poput `Number(...)`, `String(...)`, `Boolean(...)` da jasno prevedemo vrednost.

Najčešće konverzije su: - **U broj:** `Number(vrednost)` će pokušati da prosleđenu vrednost pretvori u broj. Primer: `Number("123")` vratiće broj **123** (tip number). Ako string ne može biti parsiran kao broj (npr. `"abc"`), rezultat je **NaN**. Prazan string `""` postaje 0. `true` postaje 1, `false` 0. `null` postaje 0, dok `undefined` postaje NaN (jer undefined nema numeričku reprezentaciju). - **U string:** `String(vrednost)` vraća tekstualnu reprezentaciju te vrednosti. Gotovo sve ima neku tekstualnu formu. Broj 123 postaje `"123"`. `true` postaje `"true"`. `null` -> `"null"`, `undefined` -> `"undefined"`. Ovo je slično kao `vrednost.toString()` za većinu tipova. - **U boolean:** `Boolean(vrednost)` konvertuje vrednost u `true` ili `false` po utvrđenim pravilima *truthy/falsy*. Pravilo je: **falsy** vrednosti (one koje se smatraju lažnim) su: `0`, `-0`, `""` (prazan string), `null`, `undefined`, `NaN`, i naravno `false` (koji ostaje false). Sve ostalo je **truthy** (istinitivo) – uključujući i `"0"` (string "0"), `"false"` (string), bilo koji ne-prazan string, bilo koji broj osim nule, itd. Dakle, `Boolean(0)` daje *false*, `Boolean("")` false, dok `Boolean("0")` je *true* (jer string "0" nije prazan), `Boolean([])` (prazan niz) takođe true (jer je objekat, a objekti su truthy).

Primeri konverzije:

U tabeli ispod dati su neki primeri eksplicitne konverzije korišćenjem `Number()`, `String()` i `Boolean()` funkcija:

Vrednost	`Number(v)`	`String(v)`	`Boolean(v)`
`"42"` (string)	`42` (broj)	`"42"`	**true** (nije prazan string)
`""` (prazan string)	`0`	`""`	**false** (prazan je)
`"3.14"`	`3.14` (broj)	`"3.14"`	**true** (nije prazan)
`"abc"` (nenumerički string)	`NaN`	`"abc"`	**true** (nije prazan)
`true` (boolean)	`1`	`"true"`	**true** (već true)
`false` (boolean)	`0`	`"false"`	**false** (već false)
`null`	`0`	`"null"`	**false** (null je falsy)
`undefined`	`NaN`	`"undefined"`	**false** (undefined je falsy)
`0` (broj nula)	`0`	`"0"`	**false** (0 je falsy)
`7` (broj)	`7`	`"7"`	**true** (svaki nenula broj je truthy)
`NaN`	`NaN`	`"NaN"`	**false** (NaN je falsy)

(Napomena: Konverzija `NaN` u Number ostaje `NaN` – uglavnom nema smisla raditi `Number(NaN)` ali uključeno je radi kompletnosti.)

Možete i sami isprobati ove konverzije u konzoli: recimo, ukucajte `Number("abc")` – dobićete `NaN`. Probajte `Boolean("0")` – dobićete `true`. Razumevanje ovih pravila je važno kada radite sa korisničkim unosima (koji su obično stringovi) ili podacima koji mogu biti neodređeni.

Implicitne konverzije primer: Ako napišemo `"5" - 2`, JavaScript će implicitno konvertovati `"5"` u broj 5 i rezultat će biti 3. Ali `"5" + 2` neće dati 7, već `"52"` – jer za operator **+** ako je jedan operand string, izvršiće **konkatenaciju** (spajanje stringova) umesto sabiranja. O ovome više u sledećem delu o operatorima.

Osnovni operatori nad podacima

Sada kada znamo tipove, pogledajmo osnovne **operatore** koje možemo primenjivati na te vrednosti. Najvažniji su nam aritmetički operatori za brojeve i operator konkatenacije za stringove.

- **Sabiranje (+)** – koristi se za **dvije svrhe**: sabiranje brojeva i spajanje stringova. Ako oba operanda nisu string, JavaScript će pokušati da ih shvati kao brojeve i sabere. Ali ako je makar jedan operand string, izvršiće se konkatenacija (pretvarajući drugu vrednost u string). Primer: `5 + 3` rezultira 8 (broj), dok `"Hello " + "world"` daje `"Hello world"`. Kombinovano: `"Score: " + 5` daje `"Score: 5"` (broj 5 je preveden u "5"). Oprezno sa +, jer može dovesti do neočekivanih rezultata ako tipovi nisu kako mislite.

- **Oduzimanje (-)**, **Množenje (*)**, **Deljenje (/)** – ovi operatori uglavnom rade **samo numerički**. Ako im date string operand, pokušaće da ga konvertuju u broj. Npr. `"10" - 3` postaje 7 (jer "10" -> 10). `"10" * "2"` postaje 20. Ako string ne može biti broj: `"abc" - 2` → NaN. Deljenje ima posebna dva slučaja: deljenje nulom (npr. `5/0` → Infinity, beskonačnost) i 0/0 → NaN (matematički nedefinisano).
- **Modul (%)** – ostatak pri deljenju dva broja. Primer: `10 % 3` daje 1 (jer 10 = 3*3 + 1). Ovo radi samo sa brojevima (pokušava konverziju ako je string). Koristi se često za proveru parnosti (npr. `if (x % 2 === 0) ...` znači x je paran).
- **Operator dodele (=)** – nije aritmetički operator, ali vredi pomenuti: služi da dodelimo vrednost promenljivoj. Može se kombinovati sa gore navedenim (+=, -=, *=, /=) za skraćenu dodelu.
- **Uvećanje/umanjenje (++ / --)** – dodaju ili oduzimaju 1 od numeričke promenljive. Postoje prefiksna i postfiksna verzija, ali to možemo ostaviti za kasnije detalje.
- **Logički operatori** (&&, ||, !) i **poređenja** (==, ===, <, >, itd.) ćemo obraditi u narednim lekcijama kada radimo uslove (Dan 3), tako da njih ovde nećemo detaljno.

Da vidimo neke primere operatora u praksi i njihov rezultat:

Izraz	Rezultat	Objašnjenje
`5 + 2`	`7` (Number)	Sabiranje dva broja daje broj (aritmetički zbir).
`5 - 2`	`3` (Number)	Oduzimanje dva broja.
`5 * 2`	`10` (Number)	Množenje dva broja.
`5 / 2`	`2.5` (Number)	Deljenje (uvek dobiješ broj – može biti i decimalan rezultat).
`5 % 2`	`1` (Number)	Ostatak pri deljenju 5 sa 2 je 1.
`"5" + 2`	`"52"` (String)	2 je konvertovan u "2", zatim spojen sa "5" (konkatenacija stringova).
`5 + "2"`	`"52"` (String)	Isto kao iznad – broj 5 u string, rezultat "52".
`"Hello, " + "Ana"`	`"Hello, Ana"` (String)	Konkatenacija dva stringa daje duži string.
`"5" - 2`	`3` (Number)	String "5" je konvertovan u broj 5, zatim 5-2=3.
`"10" * "2"`	`20` (Number)	Obe vrednosti su stringovi ali obe izgledaju kao brojevi pa su konvertovane u brojeve 10 i 2.
`"abc" * 3`	`NaN` (Number)	String "abc" ne može postati broj → rezultat nije broj (NaN).
`5 + true`	`6` (Number)	`true` se implicitno tretira kao 1, pa 5+1=6.
`5 + false`	`5` (Number)	`false` → 0, pa 5+0=5.
`5 + null`	`5` (Number)	`null` → 0, pa 5+0=5.
`5 + undefined`	`NaN` (Number)	`undefined` → NaN, pa rezultat NaN.
`1 / 0`	`Infinity` (Number)	Deljenje pozitivnog broja nulom daje beskonačno.

Izraz	Rezultat	Objašnjenje
`-1 / 0`	`-Infinity` (Number)	Deljenje negativnog broja nulom daje -beskonačno.

Kao što se vidi, **operator + je poseban** jer može značiti i sabiranje i spajanje stringova. Ostali aritmetički operatori uglavnom konvertuju operande u brojeve. Ove pravila su deo tzv. *type coercion* sistema u JS. **Ključno za vas kao početnika** je da budete svesni tih implicitnih konverzija kako ne biste bili iznenađeni. Recimo, veoma česta greška je:

```
let cena = "100"; // ova vrednost je možda došla kao string (npr. iz input
polja)
let kolicina = 2;
let ukupno = cena * kolicina;
console.log("Ukupno: " + ukupno); // "Ukupno: 200"
let ukupno2 = cena + kolicina;
console.log("Ukupno2: " + ukupno2); // "Ukupno2: 1002" - greška u logici!
```

U gornjem primeru, `cena * kolicina` daje očekivani broj 200 jer se `"100"` i `2` množe (string "100" → 100 brojno). Ali `cena + kolicina` ne sabira brojeve, već spaja string "100" i string "2" (jer se broj 2 konvertuje u "2") i dobijemo `"1002"`. Ovakve situacije mogu napraviti bug u programu. **Rešenje:** kada sakupljate podatke od korisnika (koji su tipično stringovi), eksplicitno konvertujte u broj pre sabiranja: npr. `Number(cena) + Number(kolicina)` da biste dobili 102 umesto "1002". Ili koristite `parseInt` / `parseFloat` ako je potrebno. Uvek razmišljajte o tipu vrednosti koju imate!

Specijalne numeričke vrednosti: `NaN` i `Infinity`

Već smo ih par puta spomenuli, ali da naglasimo:

- `NaN` (*Not a Number*): Specijalna vrednost koja označava da numerički rezultat nije definisan ili nije broj. Npr. `parseInt("blabla")` daje `NaN`, kao i rezultat 0/0 ili bilo koja nedozvoljena operacija (poput sqrt(-1) ako bismo imali takvo nešto). `NaN` je zanimljiv jer je jedinstven po tome da nije jednak ničemu, čak ni samom sebi (`NaN === NaN` je false!). Da bi proverili da li je neka vrednost NaN, koristimo funkciju `Number.isNaN(vrednost)` ili ispitujemo `isNaN(vrednost)`. Tip `NaN` je i dalje `number` (jer spada u numerički tip).
  Debug savet: Ako u programu dobijete `NaN` kao rezultat, znači da se negde desila nevalidna matematička operacija ili konverzija. Često uzrok može biti pokušaj korišćenja `undefined` ili `null` u računu, ili parsiranje stringa koji nije broj. Tragajte unazad za izvorom te vrednosti.
- `Infinity` (i `-Infinity`): Vrednost beskonačnosti. Dešava se kada podelimo broj sa 0 (pozitivan → Infinity, negativan → -Infinity), ili kada matematička operacija prelazi maksimalan opseg broja (što je retko jer JS prelazi u Infinity prilično daleko). `Infinity` se ponaša uglavnom kao matematička beskonačnost: ako nešto dodate beskonačnosti, dobijate beskonačnost, ako podelite beskonačnost sa beskonačno dobijete NaN (neodređeno), itd. `typeof Infinity` je takođe "number".
  U praksi, Infinity nam ređe pravi probleme, ali može biti indikacija greške (npr. računate nešto i dobijete Infinity znači da je verovatno neka formula imala deljenje nulom ili slično). Možete proveriti da li je konačan broj funkcijom `Number.isFinite(vrednost)`.

Male debug vežbe (uočavanje i ispravljanje grešaka)

Sada ćemo prikazati nekoliko tipičnih grešaka koje se dešavaju početnicima, kako biste naučili da ih prepoznate i ispravite. U realnom okruženju, mentor bi vam ovde dao zadatak da pronađete zašto kod ne radi i da ga popravite. Pokušaćemo to simulirati:

- **Greška u imenovanju:** JavaScript razlikuje velika i mala slova u nazivima promenljivih. Ako pogrešite u jednom slovu, dobićete `ReferenceError` da promenljiva nije definisana. Na primer:

```
let ime = "Ivana";
console.log("Ćao, " + ime + "!");   // ispravno
console.log("Ćao, " + Ime + "!");   // Greška: Ime is not defined
```

U konzoli će se pojaviti greška da `Ime` nije definisano. Lako je prevideti ovakvu grešku, naročito ako ime promenljive sadrži više reči. **Rešenje:** Pazite na konzistentnost u pisanju naziva. Ako dobijete ReferenceError, proverite da li ste svuda isto napisali naziv. ESLint takođe pomaže – on bi odmah podvukao `Ime` jer ne postoji definicija.

- **Greška nedeklarisane promenljive:** Slična situacija – zaboravili ste da deklarišete promenljivu. Npr:

```
// Namerno izostavljena deklaracija
brojac = 1;
brojac++;
console.log(brojac);
```

Ako imate uključen strict mode ili bundler/ESLint podešen na hvatanje globalnih, ovo će prijaviti grešku da `brojac` nije definisan. U slobodnom modu, JS će napraviti globalnu promenljivu implicitno (što je loše!). **Rešenje:** Uvek koristite `let` ili `const` pri definisanju promenljivih. Ako vidite da konzola kaže "is not defined", dodajte odgovarajuću deklaraciju. (U gornjem primeru: `let brojac = 1;` na početak.)

- **Greška neinicijalizovane promenljive (undefined u izrazu):** Ako pokušate da koristite promenljivu koja nema vrednost u nekoj operaciji, dobićete `undefined` u toj operaciji, što često vodi do `NaN` ili `undefined` rezultata:

```
let broj;
console.log(broj + 3); // NaN, jer 'broj' je undefined, undefined + 3 -> NaN
```

Konzola neće baciti grešku, ali ispis je `NaN` što je znak problema. U kompleksnijem kodu, ovo može proći neopaženo i kasnije dovesti do pogrešnih proračuna. **Rešenje:** Inicijalizujte promenljive pre korišćenja. Da smo uradili `let broj = 0;` ili neku validnu vrednost, problem bi bio rešen. Takođe, možete pre upotrebe proveriti da li je `broj` definisan (nije undefined).

- **Implicitna konverzija koja kvari logiku:** Primer sa sabiranjem stringa i broja smo već naveli. Debugging ovakvog problema zahteva razumevanje kako se tipovi ponašaju. Ako dobijete neočekivani string umesto broja, ili obrnuto, razmislite "odakle dolaze ove vrednosti, jesu li

možda string umesto broja?". Tipičan scenario je unos od korisnika kroz `prompt` ili formu – ti unosi su uvek stringovi, pa čak i ako korisnik ukuca broj "5", to je string `"5"`. Ako taj rezultat neposredno saberete sa brojem, dobićete konkatenaciju. **Rešenje:** kao što smo rekli, eksplicitna konverzija: npr. `let unos = prompt("Unesi broj:"); let broj = Number(unos);` zatim koristite `broj` za računanje dalje.

- **Greške sa konstantoj:** Pokušaj promene `const` vrednosti:

```
const URL = "https://example.com";
// ... kasnije
URL = "https://novi-url.com"; //   TypeError
```

Debug: Konzola će reći nešto tipa "Assignment to constant variable". To jasno ukazuje da ste pokušali promeniti konstantu. **Rešenje:** Ako vam stvarno treba promenljiva, nemojte je deklarisati kao const. Ako ne, uklonite pokušaj promene – možda je nepotrebno ili treba drugačije strukturirati kod.

- **Sintaksne greške (SyntaxError):** zaboravljeni zarez, zagrada, navodnik... Ove greške će spriječiti izvršavanje koda. Browser konzola će prijaviti tačno na kom mestu parser nije uspeo. Često VS Code sam ukaže na to (cveni ispod koda). **Rešenje:** Pažljivo čitajte poruku i pogledajte tu liniju i prethodnu – mnoge sintaksne greške su samo prosto kucanje. Recimo, poruka `Unexpected end of input` obično znači da vam fali jedan `}` negde (file se završio pre nego što je neka { zatvorena). Poruka `missing ) after argument list` znači da ste zaboravili zatvoriti zagradu funkcije ili if-a itd.

Sumirano pravilo za debugovanje: Uvek prvo pogledajte konzolu kada nešto "ne radi". Poruke greške su tu da vam pomognu. Uz malo prakse, brzo ćete ih tumačiti i pronalaziti uzrok problema.

Praktični zadaci za vežbu (Dan 2)

Pređimo sada na nekoliko **mini-zadataka** koji učvršćuju koncepte promenljivih i tipova podataka. Ove zadatke možete raditi u konzoli ili u posebnom JS fajlu koji uključite u HTML, kako vam je zgodnije:

- **Zadatak 1: Zbir dva broja.**
  Napravite dve promenljive `a` i `b`, dodelite im neke numeričke vrednosti. Izračunajte njihov zbir u novoj promenljivoj `sum` i prikažite rezultat u konzoli. Pokušajte i sa različitim brojevima. Na primer:

  ```
  let a = 5;
  let b = 7;
  let sum = a + b;
  console.log("Zbir brojeva " + a + " i " + b + " je " + sum);  //
  očekivano: "Zbir brojeva 5 i 7 je 12"
  ```

  Isprobajte promenom vrednosti a i b ili probajte sabiranje broja i stringa slučajno da vidite šta će se desiti.

- **Zadatak 2: Spajanje stringova.**
  Neka su date dve promenljive `ime` i `prezime` koje drže vaše ime i prezime. Kreirajte promenljivu `punoIme` koja će spojiti te dve uz razmak između. Ispišite rezultat. Primer:

```
let ime = "Petar";
let prezime = "Petrović";
let punoIme = ime + " " + prezime;
console.log("Puno ime:", punoIme);  // "Puno ime: Petar Petrović"
```

  Takođe, možete dodavati i druge stringove: npr. napravite pozdrav:
  `"Zdravo, " + ime + "!"`. Eksperimentišite sa konkatenacijom više stringova.

- **Zadatak 3: Deljenje nulom i NaN.**
  U konzoli proverite šta se dešava kad podelite neki broj sa nulom, npr. `10 / 0`. Očekujemo Infinity. Probajte i `0 / 0` – trebalo bi da dobijete NaN. Takođe, što dobijete za `typeof Infinity` i `typeof NaN`?

```
console.log(10 / 0);        // Infinity
console.log(-10 / 0);       // -Infinity
console.log(0 / 0);         // NaN
console.log(typeof Infinity); // "number"
console.log(typeof NaN);      // "number"
```

  Ovi rezultati pokazuju neke od specifičnosti JavaScript-a koje smo opisali. Razmislite: zašto je tip Infinity broj? (Jer spada u numeričke vrednosti.) Zašto `0/0` daje NaN – matematika to ne definiše pa ni jezik ne može.

- **Zadatak 4: Korisťenje `Math.PI`.**
  `Math` je globalni objekat koji sadrži matematičke konstante i metode. `Math.PI` je ugrađena konstanta za pi (3.14159...). Zatražite u konzoli `Math.PI` da biste videli vrednost. Zatim iskoristite je:

- **Zadatak 5: Funkcija za obim kruga.**
  Na osnovu `Math.PI`, napišite funkciju koja za dati poluprečnik računa **obim kruga**. Formula za obim je `O = 2 * π * r`. Napravite funkciju `obimKruga(r)` koja vraća tu vrednost i testirajte je sa par različitih poluprečnika. Npr:

```
function obimKruga(r) {
  return 2 * Math.PI * r;
}
console.log(obimKruga(5));  // 31.41592653589793 (što je 2 * pi * 5)
console.log(obimKruga(10)); // 62.83185307179586 (duplo veći poluprečnik
-> duplo veći obim)
```

  Ova vežba vas uvodi i u pisanje funkcija (što je tema Dana 4, ali jednostavne funkcije možemo raditi i sada). Vodite računa o tome da `Math.PI` koristite iz objekta Math, i da funkcija vraća rezultat (umesto da `console.log` unutar nje – ovde želimo da bude opšta pa da je možemo

ponovo koristiti). Takođe, iskoristite priliku da uvežbate komentarisanje: dodajte kratko objašnjenje šta funkcija radi iznad definicije, npr.

```
// Funkcija koja računa obim kruga na osnovu poluprečnika
function obimKruga(r) { ... }
```

Upoređujući vaše rešenje sa očekivanim, proverite da li sve radi. Ako nešto ne valja, upotrebite debug veštine: konzola poruke, `typeof` provere, dodavanje `console.log` unutar funkcije da vidite šta stiže itd.

Dobre navike: imenovanje, odvajanje koda i komentarisanje

Za kraj ovih uvodnih dana, skrenimo pažnju na neke **dobre prakse** koje će vam mentori i kolege uvek naglašavati:

- **Smisleno imenovanje promenljivih:** Imena promenljivih treba da odražavaju šta ta promenljiva predstavlja. Izbegavajte jednokratna slova (osim u petljama za iteratore poput `i, j`). Npr. umesto `x` ili `pom`, bolje `brojPokusaja`, `trenutnaStrana`, `korisnickoIme` itd. To čini kod samodokumentujućim. Takođe, pratite konvencije – u JavaScript-u je uobičajen **camelCase** za nazive (`mojaPromenljiva`, `brojPokusaja`), dok su npr. konstante ponekad sve velikim slovima (`MAX_COUNT`). Nikad ne stavljajte razmake ili crtice u imena (ne dozvoljava sintaksa), i ne počinjite brojkom. Koristite engleski ili srpski u imenima, kako se tim već dogovori, ali budite dosledni.
- **Odvajanje logike od prezentacije:** Već smo vežbali – JavaScript kod stavljamo u zasebne fajlove kada prelazi nekoliko linija. HTML koristimo samo za strukturu i sadržaj, CSS za stil, a JS za logiku. Ovo načelo **separacije** je ključno za čitljivost i timski rad – dizajner može da menja HTML/CSS, programer JS, bez konflikta. Čak i unutar JS koda, pokušajte da logiku razdvajate na funkcije (kada naučimo funkcije), module itd. To će doći s vremenom; za sada je dovoljno da znate da inline script je OK za „Hello World“, ali za prave projekte ide minimalan inline kod (često samo inicijalizacija ili ništa) a sav ostali JS u fajlove.
- **Komentarisanje koda:** Komentari služe da objasne **zašto** nešto radimo ili da pojasne kompleksan deo koda. U JS, komentari mogu biti jednolinijski `// ovde komentar` ili višelinijski `/* ... */`. Dobra praksa je prije funkcije napisati šta radi, pored komplikovane logike navesti korake, ili označiti `// TODO` za nešto što treba uraditi kasnije. **Loša praksa** je pisati očigledne komentare (npr. `let x = 5; // postavi x na 5` – to se vidi iz koda). Fokusirajte se na dodavanje konteksta koji nije očigledan.
  Komentare čine kod lakšim za održavanje, pogotovo kada se neko drugi uključi ili se vi vratite posle dužeg vremena. U timskom okruženju, komentari i dobra imena su znak profesionalnosti.
- **Formatiranje i stil:** Iako nije direktno pomenuto, vredi reći – održavajte uredan kod. Uvlačenja (indentacija) i razmaci znače puno za čitljivost. VS Code + Prettier ili format on save pomaže. Linter će vas voditi oko nekih stilskih stvari takođe. Uvek možete pitati mentora koji stil vodič prate, npr. AirBnB stil, StandardJS itd., i držite ga se.
- **Male commit promene (ako koristite git):** Kao junior, bićete uvedeni i u sisteme verzionisanja. Trudite se da pravite česte commit-e sa opisnim porukama. To nije direktno JS savet, ali deo “realnog okruženja” – bolje više manjih commit-ova („Implementirana funkcija obimKruga“) nego jedan ogroman commit posle 2 dana rada.

Zaključak

Kroz Dan 1 i Dan 2 pokrili smo osnove: postavili ste radno okruženje, napravili prvu stranicu sa JavaScript-om i ispisali *“Hello World”*, a zatim zaronili u promenljive i tipove podataka, što je temelj razumevanja svakog koda. Vežbali smo deklaraciju promenljivih (`let` / `const` umesto `var`), naučili razlikovati ključne tipove (brojevi, stringovi, booleans, itd.), isprobali konverzije i operacije, i videli neke specifičnosti JavaScript-a (poput `NaN` , `typeof null` greške [6]). Takođe smo istakli važnost debugovanja i dobrih navika kodiranja od samog starta.

Nastavite da eksperimentišete sa ovim primerima dok ne postanu jasni. Postavljajte pitanja sebi: “Šta ako pokušam ovo?” i proverite u konzoli. U narednim danima nadograđivaćemo ovo znanje – već sledeći, Dan 3, uvodi uslovne strukture i petlje, gde će logički tip i operatori koje smo danas obradili doći do izražaja [7] .

Resursi za Dan 2 i dalje učenje: Preporučujemo poglavlje **“Data types”** na sajtu *JavaScript.info* [8] za još primera i objašnjenja, kao i MDN članak **“JavaScript data types and data structures”** (detaljan pregled tipova) [9] . Ovi resursi će produbiti vaše razumevanje i dati dodatne uvide (npr. o mutabilnosti objekata, što ćemo pokriti uskoro). Srećno kodiranje i ne ustručavajte se da istražujete i grešite – tako se najbolje uči!

---

[1] [2] [3] [4] [5] [7] [8] [9] JavaScript Roadmap – 30 dana do Junior Full-Stack Developera.pdf
file://file-8iASufqbh12CKmERHii27C

[6] null - JavaScript | MDN
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/null