Programiranje u skriptnim jezicima (PJS)

Nositelj: doc. dr. sc. Nikola Tanković
Asistenti:

- Luka Blašković, univ. bacc. inf.
- Alesandro Žužić, univ. bacc. inf.

Ustanova: Sveučilište Jurja Dobrile u Puli, Fakultet informatike u Puli

[2] Funkcije, doseg varijabli i kontrolne strukture

Funkcije su jedan od temeljnih konstrukata u programiranju. One omogućuju grupiranje kôda u logičke cjeline koje se mogu ponovno koristiti kroz cijeli program kao i apstrakciju složenih operacija, što nam olakšava razumijevanje i održavanje kôda.

Kontrolne strukture su konstrukti u programiranju koji odlučuju o toku izvršavanja programa.

Sadržaj

1. Uvod u funkcije

Funkcije, kao što smo već spomenuli, omogućuju grupiranje kôda u logičke cjeline koje se mogu ponovno koristiti kroz cijeli program kao i apstrakciju složenih operacija, što nam olakšava razumijevanja i održavanje kôda. U JavaScriptu, funkcije ćemo deklarirati pomoću ključne riječi `function`, nakon koje slijedi:

- ime funkcije
- lista parametara funkcije, omeđena zagradama `()` i odvojena zarezima (ako ima više parametara)
- tijelo funkcije, omeđeno vitičastim zagradama `{}`

Na primjer, možemo definirati jednostavnu funkciju `kvadriraj` koja će kvadrirati broj koji joj proslijedimo kao *argument*.

```
function kvadriraj(broj) {
  return broj * broj;
}
```

Funkcija `kvadriraj` prima jedan parametar `broj` i vraća kvadrat tog broja. Ključnom riječi `return` funkcija vraća definiranu vrijednost. Ako funkcija ne vraća ništa, koristimo `return;` ili još jednostavnije izostavimo `return` naredbu.

Možemo primjetiti kako je funkcija `kvadriraj` zapravo vrlo slična matematičkoj funkciji `f(x) = x^2`. Funkcija `f` prima jedan parametar `x` i vraća kvadrat tog broja.

Ako povućemo paralelu sa `C` familijom jezika, možemo primjetiti da kod deklaracije funkcije u JavaScriptu, kao i varijabli, ne navodimo tip podataka parametara i povratne vrijednosti. Funkcija `kvadriraj` ekvivalentna je funkciji u C-u:

```
int kvadriraj(int broj) {
  return broj * broj;
}
```

Kada se izvršavaju funkcije u JavaScriptu? Funkcije u JavaScriptu se izvršavaju kada "nešto" pozove tu funkcije, primjerice to može biti:

- kada se dogodi neki događaj (eng. *event*), npr. pritisak neke tipke
- kada se pozove direktno iz Javascript kôda
- automatski (eng. ***self-invoking***)

1.1 Osnovna sintaksa funkcija

Kako smo već rekli, funkcije se deklariraju ključnom riječi `function`, nakon koje slijedi **1. ime funkcije**, **2. lista parametara** i **3. tijelo funkcije**.

Imena funkcije mogu sadržavati slova, brojeve, povlake `_` i dolar `$` znak (ista pravila vrijede kao i kod imenovanja varijabli). Imena funkcija ne smiju počinjati brojem. Kôd koji se izvršava pišemo unutar vitičastih zagrada `{}`.

```
function imeFunkcije(parametar1, parametar2, parametar3) {
  // tijelo funkcije koje obavlja neku operaciju
}
```

Zapamtimo par pojmova:

- parametri funkcije (eng. ***function parameters***) su navedeni unutar zagrada `()` u definiciji funkcije.

- argumenti funkcije (eng. ***function arguments***) su vrijednosti koje se proslijeđuju funkciji kada se ona poziva.
- najvažnije, unutar funkcije, parametri (argumenti) se ponašaju kao **lokalne varijable**.

1.2 Pozivanje funkcije

Deklariranje funkcije neće pozvati funkciju, već samo definira funkciju. Da bismo pozvali funkciju, koristimo ime funkcije, operator `()` i unutar njega argumente koje proslijeđujemo funkciji. Primjerice, kako bi pozvali našu funkciju `kvadriraj` s arugmentom `5` i ispisali rezultat u konzolu, pišemo sljedeći kôd:

```
console.log(kvadriraj(5)); // 25
```

Deklarirajmo funkciju `toCelsius` koja će pretvoriti temperaturu iz Fahrenheit u Celzijevu. Formula za pretvorbu je: `C = 5/9 * (F - 32)`.

Funkciju smo definirali ovako:

```
function toCelsius(fahrenheit) {
  return (5 / 9) * (fahrenheit - 32);
}
```

Idemo pozvati funkciju s argumentom `77` i ispisati rezultat u konzolu:

```
console.log(toCelsius(77)); // 25
```

Dobili smo rezultat `25`, odnosno 77°F je 25°C.
Što će ispisati sljedeći kôd?

```
let value = toCelsius();
console.log(value); // ?
```

Odgovor je `NaN` (eng. ***Not a Number***). Zašto? Funkcija `toCelsius` očekuje jedan argument, a mi nismo proslijedili niti jedan. Kako bismo izbjegli ovakve situacije, možemo postaviti defaultnu vrijednost za parametar funkcije, na primjer:

```
function toCelsius(fahrenheit = 0) {
  return (5 / 9) * (fahrenheit - 32);
}
```

Poziv funkcije `toCelsius()` sada će nam vratiti `0`, jer smo postavili defaultnu vrijednost za parametar `fahrenheit`.

Sada će nam `toCelsius()` vratiti `-17.777`.

JavaScript nam neće dati grešku ako slučajno pozovemo funkciju bez `()` operatora, već će to tretirati kao referencu na samu funkciju. Ovo može biti korisno u nekim situacijama, ali u pravilu želimo ovo izbjegavati.

```
let value = toCelsius;
console.log(value); // [Function: toCelsius]
```

Vježba 1

Napišite funkciju `pozdrav` koja će primati jedan argument `ime` te će ispisati poruku i vratiti string vrijednost "Pozdrav, `ime`!". Funkciju pozovite s argumentom `"Ivan"` i ispišite rezultat u konzolu. Kada to napravite dodajte defaultnu vrijednost za parametar `ime` koja će biti `"stranac"`.

Rezultat:

1.3 Funkcije možemo koristiti raznoliko

U JavaScriptu, funkcije se mogu koristiti na jednak način kao što koristimo varijable. To znači da ih možemo dodijeliti varijablama, proslijediti kao argumente drugim funkcijama, koristiti kao pridruživanje vrijednosti objektima i sl.
Primjerice, umjesto da koristimo varijablu za pohranu rezultata funkcije, možemo koristiti sam poziv funkcije!

Uzmimo našu funkciju `kvadriraj`:

```
function kvadriraj(broj) {
  return broj * broj;
}

let rezultat = kvadriraj(5);
let text = "Rezultat kvadriranja broja 5 je: " + rezultat;
console.log(text); // Rezultat kvadriranja broja 5 je: 25
```

možemo napisati i ovako:

```
let text2 = "Rezultat kvadriranja broja 5 je: " + kvadriraj(5);
console.log(text2); // Rezultat kvadriranja broja 5 je: 25
```

Što bi se dogodilo ako kôd posložimo na ovaj način?

```
let text3 = kvadriraj(5) + " je rezultat kvadriranja broja 5.";
function kvadriraj(broj) {
  return broj * broj;
}
console.log(text3); // ?
```

Primjetite da smo pozvali funkciju `kvadriraj` prije nego smo ju deklarirali. JavaScript će prvo pročitati sve deklaracije funkcija i varijabli prije nego počne izvršavati kôd, tako da ovaj kôd neće proizvesti grešku i ispisat će `25 je rezultat kvadriranja broja 5.`. Ovo ponašanje se zove **Function hoisting**.
Dakle prethodni kôd je ekvivalentan ovome:

```
function kvadriraj(broj) {
  return broj * broj;
}
let text3 = kvadriraj(5) + " je rezultat kvadriranja broja 5.";
console.log(text3); // 25 je rezultat kvadriranja broja 5.
```

Napomena, navedeno ponašanje odnosi samo na deklaracije funkcija, ne i na funkcijske izraze (eng. ***function expressions***). O funkcijskim izrazima više u nastavku skripte.

2. Doseg varijabli i funkcijski izrazi

Doseg varijabli (eng. ***variable scope***) odnosi se na pravila gdje u kôdu varijabla može biti korištena/pročitana. U JavaScriptu, varijable deklarirane unutar funkcije su **lokalne varijable** i mogu se koristiti samo unutar te funkcije. Varijable deklarirane izvan funkcije su globalne varijable i mogu se koristiti bilo gdje u kôdu (ako nisu unutar nekog drugog bloka).

```
// Kôd ovdje ne može koristiti varijablu x
function myFunction() {
  let x = 10;
  // Kôd ovdje može koristiti varijablu x
  console.log(x); // 10
}
// Kôd ovdje ne može koristiti varijablu x
console.log(x); // ReferenceError: x is not defined
```

Budući da se lokalne varijable prepoznaju samo unutar njihovih funkcija, varijable s istim imenom mogu postojati u različitim funkcijama.
Važno je napomenuti da se lokalne varijable stvaraju svaki put kada se funkcija pozove, a dealociraju kada se funkcija završi.

```
// Ove varijable definirane su u globalnom dosegu
const number_1 = 20;
const number_2 = 10;

// Ova funkcija definirana je u globalnom dosegu
function pomnozi() {
  return number_1 * number_2;
}

console.log(pomnozi()); // 200
```

Ovo je jasno, međutim hoće li sljedeći kôd ispisati `100` ili dati grešku?

```
const number_1 = 20;
const number_2 = 10;

function pomnozi() {
  const number_1 = 2;
  const number_2 = 50;
  return number_1 * number_2;
}

console.log(pomnozi()); // ?
```

▶ Odgovor

```
console.log(pomnozi()); // 100
```

2.1 Blokovski opseg (eng. *block scope*)

U JavaScriptu, varijable deklarirane s ključnim riječima `let` i `const` imaju blokovski opseg. To znači da su vidljive samo unutar bloka kôda u kojem su deklarirane, slično kao lokalne varijable deklarirane unutar funkcija, blok kôda se definira vitičastim zagradama `{}`.

```
const x = 10;
// x ovdje iznosi 10
{
  const x = 2;
  // x ovdje iznosi 2
}
// x ovdje iznosi 10
console.log(x); // 10
```

Možemo primjetiti da se varijabla `x` deklarirana unutar bloka `{}` ponaša kao lokalna varijabla unutar bloka, a varijabla `x` deklarirana izvan bloka ponaša se kao globalna varijabla.

Ponovna deklaracija varijable s ključnom riječi `let` ili redeklaracija ključnom riječi `const`, unutar istog dosega, uzrokovat će grešku!

```
let x = 10; // Okej
const x = 2; // SyntaxError: Identifier 'x' has already been declared

{
  let x = 2; // Okej
  const x = 2; // SyntaxError: Identifier 'x' has already been declared
}
{
  const x = 2; //Okej
  const x = 2; // SyntaxError: Identifier 'x' has already been declared
}
```

Uočimo i ovaj primjer: Ponovna deklaracija `const` varijable, unutar istog dosega, uzrokovat će grešku!

```
const x = 10; // Okej
x = 2; // TypeError: Assignment to constant variable.
let x = 2; // SyntaxError: Identifier 'x' has already been declared
const x = 2; // SyntaxError: Identifier 'x' has already been declared

{
  const x = 2; // Okej
  x = 2; // TypeError: Assignment to constant variable.
  let x = 2; // SyntaxError: Identifier 'x' has already been declared
  const x = 2; // SyntaxError: Identifier 'x' has already been declared
}
```

Međutim, ponovna deklaracija `const` varijable, unutar različitih dosega, neće uzrokovati grešku!

```
const x = 10; // Okej
{
  const x = 2; // Okej
}
{
  const x = "Pas"; // Okej
}
```

Kao što je već rečeno u prethodnoj skripti, varijable deklarirane s ključnom riječi `var` nemaju blokovski opseg već funkcionalni, što znači da su vidljive unutar funkcija u kojoj su deklarirane, kao i unutar svih blokova i podfunkcija. Ovo ponašanje može dovesti do neočekivanih rezultata i grešaka, stoga se preporučuje korištenje isključivo ključnih riječi `let` i `const` koje imaju blokovski opseg, umjesto `var`.

```
var x = 1;
{
  var x = 2;
}
console.log(x); // 2 - neočekivano! Zadržimo se na ključnim riječima let i const!
```

```
let x = 1;
const y = 2;
{
  let x = 2;
  const y = 3;
}
console.log(x, y); // 1 2 - očekivano!
```

Za one koji žele naučiti više o blokovskom opsegu, i function hoistingu, link je ovdje.

2.2 Ponovno deklariranje funkcija

Ponovno deklariranje funkcija u JavaScriptu s ključnom riječi `function` dozvoljeno je ovisno o dosegu gdje se funkcija deklarira.
Deklaracije funkcija sa ključnom riječi `function` ponašaju se slično kao `var` i mogu se ponovno deklarirati s još jednom `function` ili `var` deklaracijom, ali ne sa `let`, `const` ili `class` deklaracijom.

```
function a(b) {}
function a(b, c) {}
console.log(a.length); // 2 - broj parametara zadnje deklarirane funkcije
let a = 2; // SyntaxError: Identifier 'a' has already been declared
```

Ako "overridamo" funkciju s `var` deklaracijom, to će raditi, ali još jednom, nije preporučljivo.

```
var a = 1;
function a() {}
console.log(a); // 1
```

2.3 Funkcijski izrazi

Funkcijski izrazi (eng. ***function expressions***) su način definiranja funkcija kao vrijednosti varijable. Mogu se koristiti kako bi **definirali funkciju unutar izraza**.

Funkcijski izrazi također se definiraju s ključnom riječi `function`, ali se razlikuju od "deklaracija funkcija" po tome što se mogu dodijeliti varijablama, proslijediti kao argumenti drugim funkcijama, koristiti kao pridruživanje vrijednosti objektima i sl. Sintaksa je vrlo slična kao i kod klasične `function` deklaracije.

```
const izracunaj_povrsinu_pravokutnika = function (duzina, sirina) {
  return duzina * sirina;
};
console.log(izracunaj_povrsinu_pravokutnika(5, 3)); // 15 - funkcijski izraz pozivamo na
isti način kao i deklarirane funkcije
```

Kako razlikujemo deklaraciju funkcije i funkcijske izraze? Uzmimo za primjer funkciju `zbroji` koja zbraja dva broja.

Deklaracija funkcije izgleda ovako:

```
function zbroji(a, b) {
  return a + b;
}
```

Funkcijski izraz izgleda ovako:

```
const zbroji = function (a, b) {
  return a + b;
};
```

Možemo primjetiti da se kod funkcijskog izraza funkcija "izrađuje" s desne strane operatora dodjeljivanja `=`.

Kako smo ranije spomenuli, u poglavlju 1.3, **function hoisting** ponašanje dovodi do toga da se deklaracije funkcija mogu pozvati prije nego su deklarirane. Međutim, to se ne odnosi na funkcijske izraze. Funkcijski izrazi se ponašaju kao bilo koja druga varijabla, i ne mogu se pozvati prije nego su deklarirane.

```
zbroji(2, 3); // 5
function zbroji(a, b) {
  console.log(a + b);
  return a + b;
}
```

Funkcijski izraz:

```
zbroji(2, 3); // TypeError: zbroji is not a function
let zbroji = function (a, b) {
  console.log(a + b);
  return a + b;
};
```

Možemo li deklarirati funkciju unutar funkcije? Naravno 🙂

```
function vanjskaFunkcija() {
  function unutarnjaFunkcija() {
    console.log("Pozdrav iz unutarnje funkcije!");
  }
  console.log("Pozdrav iz vanjske funkcije!");
  unutarnjaFunkcija();
}
vanjskaFunkcija();
// Ispis:
// Pozdrav iz vanjske funkcije!
// Pozdrav iz unutarnje funkcije!
```

Isto tako, možemo deklarirati i funkcijski izraz unutar funkcije.

```
function vanjskaFunkcija() {
  const unutarnjaFunkcija = function () {
    console.log("Pozdrav iz unutarnje funkcije!");
  };
  console.log("Pozdrav iz vanjske funkcije!");
  unutarnjaFunkcija();
}
vanjskaFunkcija();
// Ispis:
// Pozdrav iz vanjske funkcije!
// Pozdrav iz unutarnje funkcije!
```

Svaka funkcija ima svoj svoj lokalni doseg varijabli, što znači da varijable deklarirane unutar unutarnje funkcije nisu vidljive vanjskoj funkciji (vanjska je ona koja omeđuje unutarnju)?

```
function vanjskaFunkcija() {
  const unutarnjaFunkcija = function () {
    const x = 5;
    console.log("Pozdrav iz unutarnje funkcije!");
  };
  console.log("Pozdrav iz vanjske funkcije!");
  unutarnjaFunkcija();
  console.log(x); // ReferenceError: x is not defined
}
vanjskaFunkcija();
```

Međutim, kako svaka funkcija može vratiti vrijednost putem `return` naredbe, tako unutarnja funkcija može vratiti vrijednost vanjskoj funkciji.

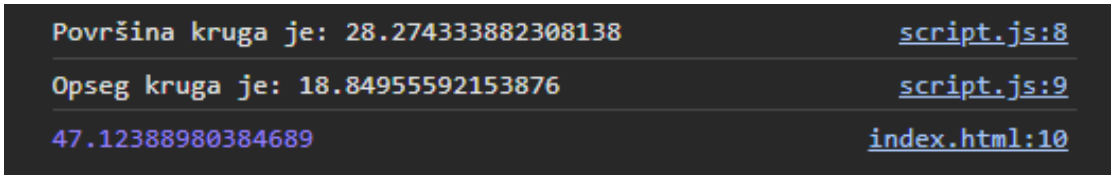
```
function vanjskaFunkcija() {
  const unutarnjaFunkcija = function () {
    return "Pozdrav iz unutarnje funkcije!";
  };
  console.log("Pozdrav iz vanjske funkcije!");
  const poruka = unutarnjaFunkcija();
  console.log(poruka); // Pozdrav iz unutarnje funkcije!
}
vanjskaFunkcija();
```

Vježba 2

Napišite funkciju `sve_o_krugu(r)` s jednim parametrom `r` koji predstavlja radijus kruga. Funkcija treba sadržavati dvije unutarnje funkcije `povrsina` i `opseg` koje će računati površinu i opseg kruga i vraćati vanjskoj funkciji rezultate. Jedna od dvije unutarnje funkcije treba koristiti funkcijski izraz, a druga deklaraciju funkcije. Vanjska funkcija treba ispisati rezultate unutarnjih funkcija u konzolu. Za vrijednost broja π koristite `Math.PI`. Vanjska funkcija treba u lokalnu varijablu `zbroj` pohraniti zbroj površine i opsega kruga i vratiti **tu vrijednost**. Rezultat funkcije `sve_o_krugu(3)` pohranite u globalnu varijablu `zbroj` te ju ispišite u konzolu.

EduCoder šifra: `krug`

Rezultat:



Vježba 3

Napišite funkciju `lessby20_others(x, y, z)` koja prima tri cjelobrojna argumenta: `x`, `y` i `z`. Funkcija treba provjeriti i vratiti `true` ako bilo koji od ovih brojeva zadovoljava sljedeće uvjete:

- Broj je veći ili jednak 20.

- Broj je manji od barem jednog od preostala dva broja.

U svim ostalim slučajevima, funkcija treba vratiti `false`.

EduCoder šifra: `lessby20_others`

Rezultat:

```
console.log(lessby20_others(23, 45, 10)); //true
console.log(lessby20_others(23, 23, 10)); //false
console.log(lessby20_others(10, 25, 75)); //true
```

3. Uvod u paradigmu funkcijskog programiranja

Funkcijsko programiranje (eng. ***functional programming***) je paradigma programiranja koja se temelji na korištenju funkcija kao osnovnih gradivnih blokova.
Funkcijsko programiranje možemo zamisliti kao princip pisanja računalnih programa gdje primarno koristimo funkcije kao osnovne gradivne blokove, a ne npr. objekte, klase, varijable i sl.

Funkcijsko programiranje možemo usporediti s LEGO kockicama. Svaki LEGO blok (funkcija) ima svoju specifičnu svrhu i obavlja jednu stvar dobro. Kombiniranjem tih blokova možemo izgraditi složene strukture (programe).

3.1 Čiste funkcije

Jedno od svojstava funkcijskog programiranja je **čista funkcija** (eng. ***pure function***). Čista funkcija je funkcija koja ne mijenja stanje varijabli izvan svojeg dosega, ali niti ne ovisi o stanju varijabli izvan svog dosega. Čista funkcija uvijek vraća isti rezultat za iste ulazne parametre.

```
// Čista funkcija - ne ovisi o stanju varijabli izvan svog dosega i ne mijenja stanje
varijabli izvan svog dosega
function kvadriraj(broj) {
  return broj * broj;
}
console.log(kvadriraj(5)); // 25
```

```
// Nečista funkcija - ovisi o stanju varijabli izvan svog dosega i mijenja stanje
varijabli izvan svog dosega
let rezultat = 0;
let broj = 5;
function kvadriraj() {
  rezultat = broj * broj;
  return rezultat;
}
console.log(kvadriraj()); // 25
```

3.2 Imutabilnost

Imutabilnost odnosno nepromjenjivost (eng. ***immutability***) je još jedno svojstvo funkcijskog programiranja. Imutabilnost se odnosi na to da se vrijednosti varijabli ne mijenjaju jednom nakon što su definirane. Uzmimo za primjer inkrement/dekrement operatore `++` i `--` koji mijenjaju vrijednosti varijable nad kojom se koriste. U funkcijskom programiranju, umjesto da mijenjamo vrijednost varijable, htjeli bismo stvoriti novu varijablu s novom vrijednošću.

```
let x = 5;
x++; // mijenja vrijednost varijable x
console.log(x); // 6
```

```
let x = 5;
let y = x + 1; // stvara novu varijablu y s novom vrijednošću
console.log(x, y); // 5 6
```

ili

```
function inkrement(x) {
  return x + 1;
}
let x = 5;
let y = inkrement(x); // stvara novu varijablu y s novom vrijednošću
console.log(x, y); // 5 6
```

3.3 Funkcije višeg reda

Funkcije višeg reda (eng. ***higher-order functions***) su funkcije koje primaju druge funkcije kao argumente ili vraćaju druge funkcije kao rezultat. Funkcije višeg reda omogućuju nam da apstrahiramo zajedničke obrasce u funkcijama i da ih koristimo kao argumente drugim funkcijama.

Idemo napraviti jednostavan kalkulator koji može zbrajati i oduzimati. Napišimo funkcije `zbroji` i `oduzmi` koje će primati dva argumenta i vraćati rezultat zbrajanja i oduzimanja.

```
function zbroji(a, b) {
  return a + b;
}

function oduzmi(a, b) {
  return a - b;
}

console.log(zbroji(5, 3)); // 8
console.log(oduzmi(5, 3)); // 2
```

Rekli smo da je funkcija višeg reda koja prima drugu funkciju kao arugment ili vraća drugu funkciju kao rezultat. Možemo implementirati funkciju `izracunaj` koja će primiti funkciju `operacija` i dva broja `a` i `b` te će vratiti rezultat funkcije `operacija` s argumentima `a` i `b`.

```
function izracunaj(operacija, a, b) {
  return operacija(a, b);
}
console.log(izracunaj(zbroji, 5, 3)); // 8
console.log(izracunaj(oduzmi, 5, 3)); // 2
```

Želimo deklarirati funkcije `double` i triple koje će primati jedan broj i vraćati dvostruko odnosno trostruko veći broj.

```
function double(x) {
  return x * 2;
}

function triple(x) {
  return x * 3;
}

console.log(double(5)); // 10
console.log(triple(5)); // 15
```

Što ako želimo dodati funkcije `quadruple` i `quintuple` koje će vraćati četverostruko odnosno petostruko veći broj? Recimo da želimo ostati na tome da naša funkcija prima samo jedan argument. Možemo li to riješiti pomoću funkcija višeg reda?
Možemo! Deklarirati ćemo funkciju `mutliplier` koja će primati jedan argument `multiplier` te će vraćati funkciju koja će primati jedan argument `x` i vraćati `x * multiplier`.

Dakle `multiplier` je funkcija višeg reda jer vraća funkciju kao povratnu vrijednost.

```
function multiplier(value) {
  return function (x) {
    return x * value;
  };
}

let double = multiplier(2);
let triple = multiplier(3);
let quadruple = multiplier(4);
let quintuple = multiplier(5);

console.log(double(5)); // 10
console.log(triple(5)); // 15
console.log(quadruple(5)); // 20
console.log(quintuple(5)); // 25
```

Samostalni zadatak za vježbu 2

Napomena: Ne predaje se i ne boduje se. Zadatak možete i ne morate rješavati u EduCoder aplikaciji.

EduCoder šifra: `bmi_and_heron`

1. Napišite **funkciju** `provjera_parnosti` koja će provjeravati je li broj paran ili neparan. Funkcija treba primiti jedan parametar `broj` i vratiti boolean vrijednosti "true" za parnost ili "false" za neparnost. Funkciju napišite **bez** upotrebe selekcija (if, else, switch) Funkciju pozovite s argumentom `5` i ispišite rezultat u konzolu.
2. Napišite **funkcijski izraz** `izrazunaj_povrsinu` koji računa površinu pravokutnika. U varijablu `povrsina` pohranite taj funkcijski izraz. Ispišite vrijednost `povrsina(8,6)` u konzolu.
3. Napišite **funkcijski izraz** `BMI` koji računa BMI (Body Mass Index) osobe. BMI se računa prema formuli `BMI = težina / (visina * visina)`. Ispišite u konzolu BMI osobe koja ima težinu 75 kg i visinu 1.75 m.
4. Napišite **funkciju** `heron()` koja će računati površinu trokuta prema Heronovoj formuli. Funkcija treba primiti tri parametra `a`, `b` i `c` koji predstavljaju duljine stranica trokuta.
    - Heronova formula: `P = √(p * (p - a) * (p - b) * (p - c))` gdje je `p` poluopseg trokuta, a računa se prema formuli `p = (a + b + c) / 2`. Koristite funkciju `Math.sqrt()` za računanje korijena (Sintaksa je: `Math.sqrt(broj)`)
    - Napišite funkcijski izraz `poluopseg` koji će primiti tri parametra `a`, `b` i `c` te vratiti poluopseg trokuta prema danoj formuli. Funkcijski izraz mora biti definiran unutar funkcije `heron()`.
    - Unutar funkcije Heron, deklarirajte novu konstantu `p` koja će pohraniti vrijednost funkcijskog izraza `poluopseg(a, b, c)`.
    - Rezultat funkcije `heron(3, 4, 5)` pohranite u varijablu `povrsina_trokuta` te ispišite u konzolu: `Trokut s duljinama stranica 3, 4 i 5 ima površinu: povrsina_trokuta(3, 4, 5) cm2` koristeći `template_literals`.
5. Sljedeći JavaScript kôd sadrži nekoliko grešaka. Pronađite i ispravite greške kako bi kôd radio ispravno. Provjerite s pozivom funkcije `izracunaj(x, y, z);` koji mora ispisati `17` i `3`.

```
const x = 10;
const y = 5;
const z = 2;

function izracunaj(x, y, z) {
  let x = 5;
  let y = 3;
  let z = 2;

  function = zbroji() {
    return x + y + z;
  }
  console.log(function(zbroji(x,y,z)))

  const oduzmi = function () = {
    return y - x - z;
  }
  console.log(oduzmi());
}
// Provjera: izracunaj(x, y, z); mora ispisati sljedeće:
// 17
// 3
```

3. Kontrolne strukture

Kontrolne strukture su konstrukti koji odlučuju o toku izvršavanja programa na temelju određenih uvjeta. Ako je uvjet ispunjen tada se izvršava određeni blok radnji, inače će se izvršavati drugi blok radnji koji zadovoljava taj uvjet. Kontrolne strukture možemo podijeliti u dvije kategorije:

1. Selekcije (eng. ***Conditional statements***) - odlučuju o toku izvršavanja bloka kôda na temelju logičkog izraza koji se evaluira u `true` ili `false`.
2. Iteracije/Petlje (eng. ***Iterations***) - omogućuju izvršavanje bloka kôda više puta dok se ne ispuni uvjet definiran logičkim izrazom, koji se evaluira u `true` ili `false`.

3.1 Selekcije (eng. *Conditional statements*)

U JavaScriptu, kao i u većini programskih jezika, selekcije se pišu pomoću ključnih riječi `if`, `else if` i `else` te `switch`. Kada koristimo koju selekciju ovisi o tome koliko uvjeta želimo provjeriti:

- `if` selekciju koristimo kako bi specificirali blok kôda koji se izvršava ako je evaluirani logički izraz `true`
- `else` selekciju koristimo kako bi specificirali blok kôda koji se izvršava ako je evaluirani logički izraz `false`
- `else if` selekciju koristimo kako bi provjerili novi logički izraz ako je prethodni izraz unutar `if` ili `if else` bio `false`
- `switch` selekciju koristimo kada imamo puno alternativnih uvjeta (logičkih izraza) koje želimo provjeriti

3.1.1 `if` selekcija

Koristimo `if` selekciju kako bi specificirali blok kôda koji se izvršava ako je evaluirani logički izraz `true`. Sintaksa je sljedeća:

```
if (logicki_izraz) {
  // blok kôda koji se izvršava ako je logicki_izraz = true
}
```

Pripazite da je blok kôda uvučen unutar vitičastih zagrada `{}`. Ako je logički izraz `true`, izvršava se blok kôda unutar vitičastih zagrada `{}`. Ako je logički izraz `false`, blok kôda se preskače. Primjer:

```
let x = 10;
if (x < 5) {
  console.log("x je veći od 5"); // neće se ispisati
}
```

Ako izostavimo vitičaste zagrade `{}`, JavaScript će izvršiti samo prvu liniju kôda nakon `if` selekcije. Ovo ponašanje može dovesti do neočekivanih rezultata i grešaka, stoga se preporučuje korištenje vitičastih zagrada `{}`.

3.1.2 `else` selekcija

Koristimo `else` selekciju kako bi specificirali blok kôda koji se izvršava ako je evaluirani logički izraz `false`. Sintaksa je sljedeća:

```
if (logicki_izraz) {
  // blok kôda koji se izvršava ako je logicki_izraz = true
} else {
  // blok kôda koji se izvršava ako je logicki_izraz = false
}
```

Primjer:

```
let x = 10;
if (x < 5) {
  console.log("x je manji od 5"); // neće se ispisati
} else {
  console.log("x je veći ili jednak 5"); // ispisat će se
}
```

3.1.3 `else if` selekcija

Koristimo `else if` selekciju kako bi provjerili novi logički izraz ako je prethodni bio `false`. Sintaksa je sljedeća:

```
if (logicki_izraz_1) {
  // blok kôda koji se izvršava ako je logicki_izraz_1 = true
} else if (logicki_izraz_2) {
  // blok kôda koji se izvršava ako je logicki_izraz_2 = true
} else {
  // blok kôda koji se izvršava ako su svi prethodni logicki izrazi (logicki_izraz_1 &&
logicki_izraz_2) = false
}
```

Primjer:

```
let x = 10;
if (x < 5) {
  console.log("x je manji od 5"); // neće se ispisati
} else if (x === 5) {
  console.log("x je jednak 5"); // neće se ispisati
} else {
  console.log("x je veći od 5"); // ispisat će se
}
```

3.1.4 `switch` selekcija

`switch` selekcija koristi se kada imamo puno alternativnih uvjeta (logičkih izraza) koje želimo provjeriti. Selekcija se sastoji od ključnih riječi `switch`, `case` i `default`, gdje `switch` predstavlja izraz koji se provjerava, `case` predstavlja moguće vrijednosti izraza, a `default` predstavlja blok kôda koji se izvršava ako niti jedan od prethodnih uvjeta nije ispunjen. Sintaksa je sljedeća:

```
switch (izraz) {
  case vrijednost_1:
    // blok kôda koji se izvršava ako je izraz = vrijednost_1
    break;
  case vrijednost_2:
    // blok kôda koji se izvršava ako je izraz = vrijednost_2
    break;
  default:
  // blok kôda koji se izvršava ako niti jedan od prethodnih uvjeta nije ispunjen
}
```

Kao i u C jezicima, nakon svakog bloka kôda u `case` selekciji koristimo ključnu riječ `break` kako bi prekinuli izvršavanje selekcije. Ako izostavimo `break` naredbu, JavaScript će izvršiti sve blokove kôda nakon prvog koji zadovoljava uvjet, što može dovesti do neočekivanih rezultata i grešaka, stoga se gotovo uvijek koristi `break` naredba.

Primjer:

```
let dan = "srijeda";
switch (dan) {
  case "ponedjeljak":
    console.log("Danas je ponedjeljak");
    break;
  case "utorak":
    console.log("Danas je utorak");
    break;
  case "srijeda":
    console.log("Danas je srijeda");
    break;
  case "četvrtak":
    console.log("Danas je četvrtak");
    break;
  case "petak":
    console.log("Danas je petak");
    break;
  default:
    console.log("Vikend je!");
}
```

3.2 Selekcije s logičkim operatorima

Selekcije s logičkim operatorima koriste se kako bi provjerili više uvjeta istovremeno. U JavaScriptu, kao i u većini programskih jezika, primarno koristimo logičke operatore `&&` (i), `||` (ili) i `!` (negacija) kako bi provjerili više uvjeta istovremeno. Logički operatori vraćaju `true` ili `false` ovisno o rezultatu provjere uvjeta.

Najlakše je objasniti logičke operatore kroz konkretne primjere:

Primjer 1 - Selekcija vremena u danu (operator `&&` + `if-else` selekcija)

Prije nego što krenemo sa samim kôdom, zapisat ćemo nekoliko tvrdnji koje ćemo provjeravati logičkim operatorima:

- Ako je vrijeme između 6 i 12 sati, pozdravit ćemo s "Dobro jutro!"
- Ako je vrijeme između 12 i 18 sati, pozdravit ćemo s "Dobar dan!"
- Inače ćemo pozdraviti s "Dobra večer!"

Idemo prvo ugrubo definirati strukturu kôda:

```
let sat = 10;

if (uvjet) {
  izraz;
} else if (drugiUvjet) {
  izraz;
} else {
  izraz;
}
```

Krenimo s popunjavanjem onim redoslijedom kako smo naveli tvrdnje:
Prvi uvjet: Ako je vrijeme između 6 i 12 sati, pozdravit ćemo s "Dobro jutro!" - `if (sat >= 6 && sat < 12)` - koristimo logički operator `&&` (i) kako bi provjerili oba uvjeta istovremeno. Ako je `sat` veći ili jednak `6` i manji od `12`, odnosno, `(6 <= sat < 12)` ispisat ćemo `"Dobro jutro!"`.

```
let sat = 10;
if (sat >= 6 && sat < 12) {
  console.log("Dobro jutro!");
} else if (drugiUvjet) {
  izraz;
} else {
  izraz;
}
```

Nastavljamo dalje, drugi uvjet: Ako je vrijeme između 12 i 18 sati, pozdravit ćemo s "Dobar dan!" - `else if (sat >= 12 && sat < 18)` - koristimo logički operator `&&` (i) kako bi provjerili oba uvjeta istovremeno. Ako je `sat` veći ili jednak `12` i manji od `18`, odnosno, `(12 <= sat < 18)` ispisat ćemo `"Dobar dan!"`.

```
let sat = 10;
if (sat >= 6 && sat < 12) {
  console.log("Dobro jutro!");
} else if (sat >= 12 && sat < 18) {
  console.log("Dobar dan!");
} else {
  izraz;
}
```

I na kraju, treći uvjet: Inače ćemo pozdraviti s "Dobra večer!" - `else` - ako niti jedan od prethodnih uvjeta nije ispunjen, ispisat ćemo `"Dobra večer!"`.

```
let sat = 10;
if (sat >= 6 && sat < 12) {
  console.log("Dobro jutro!");
} else if (sat >= 12 && sat < 18) {
  console.log("Dobar dan!");
} else {
  console.log("Dobra večer!");
}
```

Primjer 2 - Provjera prihvatljivosti za zajam (operator `||`, `&&` + `if-else` selekcija)

U ovom primjeru simulirati ćemo provjeru prihvatljivosti klijenta za zajam temeljem nekoliko kriterija, koristeći logičke operatore `||` (ili) i `&&` (i).
Izmislit ćemo nekoliko tvrdnji koje ćemo provjeravati logičkim operatorima:

- Ako je klijent zaposlen i ima stabilne prihode veće od 7000 novčanih jedinica, može dobiti zajam.
- Ako je klijent samostalni obrtnik ili ima visoku kreditnu ocjenu, može dobiti zajam.
- Ako klijent ima barem 2 godine radnog iskustva ili je stariji od 25 godina i ima mjesečne prihode iznad 5000 novčanih jedinica, može dobiti zajam.

Svaka od tvrdnji je neovisna o drugima, odnosno barem jedna mora biti ispunjena kako bi klijent bio prihvatljiv za zajam!

Koje varijable možemo iščitati iz ovih tvrdnji?

- zaposlen - `boolean`
- obrtnik - `boolean`
- kreditnaOcjenaVisoka - `boolean`
- godineRadnogIskustva - `number`
- dob - `number`
- mjesecniPrihodi - `number`

Krenimo s popunjavanjem onim redoslijedom kako smo naveli tvrdnje:

Prvi uvjet: Ako je klijent zaposlen i ima stabilne prihode veće od 7000 novčanih jedinica, može dobiti zajam - `if (zaposlen == true && mjesecniPrihodi > 7000)` - koristimo logički operator `&&` (i) kako bi provjerili oba uvjeta istovremeno. Ako je `zaposlen` i `mjesecniPrihodi` veći od `7000`, odnosno, `(zaposlen == true && mjesecniPrihodi > 7000)` klijent može dobiti zajam.

```
let zaposlen = true;
let mjesecniPrihodi = 8000;

if (zaposlen == true && mjesecniPrihodi > 7000) {
  console.log("Čestitamo! Možete dobiti zajam!");
} else {
  console.log("Nažalost, ne možete dobiti zajam.");
}
```

Prisjetimo se kratko kako JavaScript evaluira tvrdnje (eng. ***expressions***) unutar kontrolnih struktura.
Što će vratiti (u što će se evaluirati), u kôdu iznad, izraz `zaposlen == true`? Odgovor je `true`.
Ako smo sigurni da je varijabla `zaposlen` uvijek tipa `boolean`, možemo izostaviti `== true` i napisati samo `if (zaposlen && mjesecniPrihodi > 7000)`.

```
let zaposlen = true;
let mjesecniPrihodi = 8000;

if (zaposlen && mjesecniPrihodi > 7000) {
  //Ovakav zapis je dovoljan, pa i čitljiviji
  console.log("Čestitamo! Možete dobiti zajam!");
} else {
  console.log("Nažalost, ne možete dobiti zajam.");
}
```

Drugi uvjet: Ako je klijent samostalni obrtnik ili ima visoku kreditnu ocjenu, može dobiti zajam - `else if (obrtnik || kreditnaOcjenaVisoka)` - koristimo logički operator `||` (ili) kako bi provjerili jedan od dva uvjeta. Ako je `obrtnik` ili `kreditnaOcjenaVisoka` istinita tvrdnja, odnosno, `(obrtnik || kreditnaOcjenaVisoka)` klijent može dobiti zajam. Primjetite da smo izostavili `== true` jer su `obrtnik` i `kreditnaOcjenaVisoka` tipa `boolean`.

```
let zaposlen = true;
let mjesecniPrihodi = 8000;

let obrtnik = true;
let kreditnaOcjenaVisoka = false;

if ((zaposlen && mjesecniPrihodi > 7000) || obrtnik || kreditnaOcjenaVisoka) {
  console.log("Čestitamo! Možete dobiti zajam!");
} else {
  console.log("Nažalost, ne možete dobiti zajam.");
}
```

Treći uvjet: Ako klijent ima barem 2 godine radnog iskustva ili je stariji od 25 godina i ima stabilne mjesečne prihode, može dobiti zajam - `(godineRadnogIskustva >= 2 || (dob > 25 && mjesecniPrihodi > 5000))` - koristimo logički operator `||` (ili) kako bi provjerili jedan od dva uvjeta. Ako je `godineRadnogIskustva` veće ili jednako `2` ili je `dob` veći od `25` i `mjesecniPrihodi` veći od `5000`, odnosno, `(godineRadnogIskustva >= 2 || (dob > 25 && mjesecniPrihodi > 5000))` klijent može dobiti zajam.

```
let zaposlen = true;
let mjesecniPrihodi = 8000;

let obrtnik = true;
let kreditnaOcjenaVisoka = false;

let godineRadnogIskustva = 3;
let dob = 28;

if (
  (zaposlen && mjesecniPrihodi > 7000) ||
  obrtnik ||
  kreditnaOcjenaVisoka ||
  godineRadnogIskustva >= 2 ||
  (dob > 25 && mjesecniPrihodi > 5000)
) {
  console.log("Čestitamo! Možete dobiti zajam!");
} else {
  console.log("Nažalost, ne možete dobiti zajam.");
}
```

Kako su uvjeti neovisni jedan o drugome, odnosno barem jedan uvjet mora biti ispunjen, možemo kompliciran izraz unutar `if` selekcije podijeliti u više manjih izraza kako bi kôd bio čitljiviji.

```
// Varijable ostaju iste

if (zaposlen && mjesecniPrihodi > 7000) {
  console.log("Čestitamo! Možete dobiti zajam!");
} else if (obrtnik || kreditnaOcjenaVisoka) {
  console.log("Čestitamo! Možete dobiti zajam!");
} else if (godineRadnogIskustva >= 2 || (dob > 25 && mjesecniPrihodi > 5000)) {
  console.log("Čestitamo! Možete dobiti zajam!");
} else {
  console.log("Nažalost, ne možete dobiti zajam.");
}
```

Vježba 4

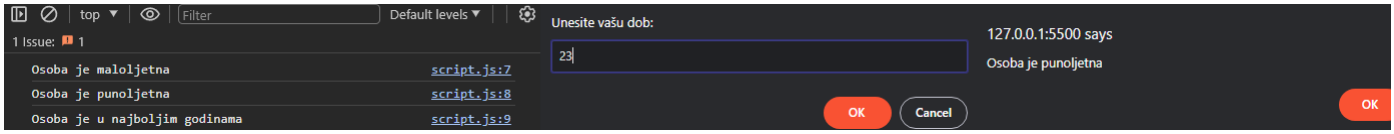
Napiši funkciju `provjeriDob(dob)` koja vraća poruku ovisno o dobi korisnika. Za dob manju od `18` godina, funkcija vraća poruku `"Osoba je maloljetna."`. Za dob između 18 i 65 godina, funkcija vraća poruku `"Osoba je punoljetna."`. Za dob veću od `65` godina, funkcija vraća poruku `"Osoba je u zlatnim godinama."`. Pozovite `provjeriDob(15)`, `provjeriDob(25)` i `provjeriDob(70)` te ispišite rezultate u konzolu. Kada to napravite, umjesto da ručno mjenjate dob, koristite `prompt` funkciju kako bi korisnik unio

dob, sintaksa je sljedeća: `let x = prompt(text, defaultText);`, gdje je `text` poruka koja se prikazuje korisniku, a `defaultText` je opcionalni argument koji predstavlja zadani tekst u polju za unos. Kada to napravite, zamjenite `console.log` sa `alert` funkcijom, sintaksa je sljedeća: `alert(poruka);`, gdje je `poruka` poruka koja se prikazuje korisniku.

EduCoder šifra: `zlatne_godine`

Napomena za EduCoder: U trenutnoj verziji EduCodera v1.4 možete pisati `prompt` i `alert` funkcije u JS dijelu editoru, no možete koristiti samo jednom. Ako želite više, preostale morate zakomentirati. U tom slučaju, preporuka je ovdje ugasiti automatsku evaluaciju i evaluirati kod ručno koristeći `CTRL/CMD + Enter`.

Rezultat:



3.3 Iteracije/Petlje (eng. *Iterations/Loops*)

Petlje su konstrukti koji omogućuju izvršavanje bloka kôda više puta dok se ne ispuni uvjet definiran logičkim izrazom. U JavaScriptu, kao i u većini programskih jezika, petlje se ostvaruju pomoću ključnih riječi `for` i `while`.

Petlje su korisne kada želimo određeni dio koda izvršavati više puta, svaki put s različitim ulaznim podacima. Na primjer, kada želimo ispisati brojeve od `1` do `10`, možemo koristiti petlju umjesto da svaki broj ispisujemo ručno.

```
console.log(1);
console.log(2);
console.log(3);
console.log(4);
console.log(5);
console.log(6);
console.log(7);
console.log(8);
console.log(9);
console.log(10);
```

možemo napisati jednostavno:

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

Postoji više vrsta `for` petlji u JavaScriptu, ali u pravilu sve rade istu stvar - ponavljaju radnju određeni broj puta (ili nijednom). Koju petlju koristimo zaključujemo ovisno o: ulaznim podacima, početku i kraju petlje, te koracima. Ova `for` petlja slična je `for` petljama u C i Java jezicima.

3.3.1 Klasična `for` petlja

Klasična `for` petlja koristi se kada znamo koliko puta želimo ponoviti blok kôda. Sastoji se od `initialization`, `condition` i `afterthought`. Sintaksa je sljedeća:

```
for (initialization; condition; afterthought) {
  statement; // blok kôda koji se izvršava dok je uvjet = true
}
```

1. `initialization` - izvršava se jednom prije početka petlje, ako postoji. Često inicijalizira varijable koje se koriste u petlji, npr. `let i = 0`, ali sintaksa dozvoljava bilo koji izraz.
2. `condition` izraz se evaluira prije svakog ponavljanja petlje. Ako je `true`, petlja i egezekucija `statement` izraza se nastavlja. Ako je `false`, petlja se prekida.
3. `statement` izraz se izvršava svaki put kada je `condition` = `true`.
4. `afterthought` izraz se izvršava nakon svakog ponavljanja petlje, ako postoji. Često se koristi za inkrementiranje ili dekrementiranje varijabli, npr. `i++`, ali sintaksa dozvoljava bilo koji izraz.

Primjer, želimo ispisati brojeve od `1` do `10`:

```
for (let i = 1; i <= 10; i++) {
  console.log(i); // ispisuje brojeve od 1 do 10 -> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
}
```

Možemo i za nazad:

```
for (let i = 10; i >= 1; i--) {
  console.log(i); // ispisuje brojeve od 10 do 1 -> 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
}
```

Kako možemo upotrijebiti `for` petlju za ispis svih parnih brojeva od `1` do `10`?

```
for (let i = 2; i <= 10; i += 2) {
  console.log(i); // ispisuje parne brojeve od 1 do 10 -> 2, 4, 6, 8, 10
}
```

Kako smo rekli da `initialization` dozvoljava bilo koji izraz pa i prazan, možemo koristiti `for` petlju i na sljedeće načine:

```
let i, j;
for (i = 0, j = 1; i < 10; i++, j++) {
  console.log(`${i} je manji za 1 od ${j}.`);
}
```

Uočite da smo varijable `i` i `j` inicijalizirali izvan petlje, ali smo ih koristili unutar petlje. Međutim, varijable je moguće deklarirati i unutar petlje, u tom slučaju ćemo ključnu riječ `let` koristiti samo jednom.

```
for (let i = 0, j = 1; i < 10; i++, j++) {
  console.log(`${i} je manji za 1 od ${j}.`);
}
```

Možemo pustiti `initialization` prazan, ali moramo imati `;` separator.

```
let i = 0;
for (; i < 10; i++) {
  console.log(i);
}
```

Izostavljanjem nekih od dijelova `for` petlje, možemo dobiti beskonačnu petlju.
Oprez, beskonačne petlje često dovode do crashanja web preglednika ili vaše aplikacije koja izvodi JavaScript kôd. Poželjno je izbjegavati beskonačne petlje.

```
// Navedene petlje će vrlo vjerojatno srušiti vaš web preglednik
for (;;) {
  console.log("Beskonačna petlja!"); // Nema inicijalizacije, uvjeta niti afterthoughta
}

for (let i = 0; ; i++) {
  console.log(i); // Nema uvjeta za prekid petlje
}

for (let i = 0; i < 10; ) {
  console.log(i); // Nema afterthoughta, petlja će beskonačno ispisivati 0
}
```

Primjer 3 - Ispis ispis brojeva od 1 do 100 koji su djeljivi s 3

Izračunajte sumu svih brojeva od `1` do `100` koji su djeljivi s 3. Koristite `for` petlju. Ovaj zadatak zahtjeva korištenje petlje za iteriranje kroz brojeve od 1 do 100, uvjetne izjave za provjeru je li broj djeljiv s 3 i varijablu za praćenje ukupne sume.

Prvo ćemo napisati kôd koji ispisuje sve brojeve od `1` do `100`.

```
for (let i = 1; i <= 100; i++) {
  console.log(i);
}
```

Dodat ćemo provjeru je li broj djeljiv s 3. To radimo s operatorom `%` koji vraća ostatak dijeljenja dva broja. Ako je ostatak dijeljenja nekog broja s `3` jednak `0`, to znači da je broj djeljiv s 3.

```
for (let i = 1; i <= 100; i++) {
  if (i % 3 === 0) {
    console.log(i);
  }
}
```

Konačno, dodat ćemo varijablu `suma` koja će pohraniti sumu svih brojeva od `1` do `100` koji su djeljivi s 3.

```
let suma = 0;
for (let i = 1; i <= 100; i++) {
  if (i % 3 === 0) {
    console.log(i);
    suma += i;
  }
}
console.log(suma); // ispisuje sumu svih brojeva od 1 do 100 koji su djeljivi s 3 -> 1683
```

3.3.2 `while` petlja

`while` petlja koristi se kada u pravilu ne znamo koliko puta želimo ponoviti blok kôda. Sastoji se od `condition`. Sintaksa je sljedeća:

```
while (condition) {
  statement; // blok kôda koji se izvršava dok je uvjet = true
}
```

Ako je `condition` = `true`, izvršava se `statement`. Ako je `condition` = `false`, petlja se prekida. Kao i kod `for` petlje, `statement` izraz se izvršava svaki put kada je `condition` = `true`.

`condition` se evaluira prije statement izraza, stoga je moguće da se `statement` izraz nikada ne izvrši ako je condition = `false`.

Primjer, sljedeća petlja će iterirati dokle god je `n` manji od 3. Primjetite da u ovom slučaju, `n` mora biti deklariran izvan petlje.

```
let n = 0;
let x = 0;
while (n < 3) {
  n++;
  x += n;
}
```

Sa svakom iteracijom, `n` se inkrementira za `1` i dodaje se na `x`. Kada je `n` = `3`, petlja se prekida. Tako da će se izvršiti `3` puta, a `x` i `n` će biti:

1. prolazak: `n` = `1`, `x` = `1`
2. prolazak: `n` = `2`, `x` = `3`
3. prolazak: `n` = `3`, `x` = `6`

Već smo rekli da beskonačne petlje želimo izbjegavati. Moramo osigurati da uvjet u `while` petlji kad tad postane `false`. Ako uvjet nikad ne postane `false`, petlja će se izvršavati beskonačno. Na primjer, sljedeća petlja će se izvršavati beskonačno:

```
while (true) {
  console.log("Beskonačna petlja!");
}
```

Dalje, pogledajmo sljedeći primjer:

```
let i = 0;
while (i < 10) {
  let text = "";
  text += "Broj " + i;
  i++;
  console.log(text); // ispisuje "Broj 0", "Broj 1", "Broj 2", "Broj 3", "Broj 4", "Broj 5", "Broj 6", "Broj 7", "Broj 8", "Broj 9"
}
```

Primjetimo da je varijabla `text` deklarirana unutar petlje. To znači da će se svaki put kada se petlja izvrši, varijabla `text` ponovno inicijalizirati. Kod petlji vrijede ista pravila o dosegu varijabli kao i kod funkcija - varijabla deklarirana unutar petlje neće biti dostupna izvan petlje.

Što ako je `i` = `11`? Petlja se neće izvršiti niti jednom, jer je uvjet `i < 10` odmah `false`. Kako bismo ispisali "Broj 10", možemo koristiti varijantu `while` petlje - `do-while` petlju.

3.3.2.1 `do-while` petlja

`do-while` petlja koristi se kada želimo da se blok kôda izvrši barem jednom, a zatim se ponavlja dok je uvjet = `true`. Sastoji se od `condition`. Sintaksa je sljedeća:

```
do {
  statement; // blok kôda koji se izvršava barem jednom, a zatim se ponavlja dok je uvjet = true
} while (condition);
```

Prebacimo prethodni primjer u `do-while` petlju. Možemo primjetiti da se `statement` blok izvrši točno jednom, budući da je uvjet `i < 10` odmah `false`.

```
let i = 11;
do {
  let text = "";
  text += "Broj " + i;
  i++;
  console.log(text); // ispisuje "Broj 11"
} while (i < 10);
```

`do-while` petlja ima svoje prednosti, ali se u praksi koristi rjeđe od `for` i `while` petlji.

3.3.3 Prekidanje petlji - `break` | `continue`

Kako bismo "naglo" prekinuli izvršavanje petlje, koristimo ključnu riječ `break`. Kada se `break` naredba izvrši, petlja se prekida i izvršavanje se nastavlja s prvim redom kôda nakon petlje. Na primjer, želimo prekinuti petlju kada dođemo do broja `15` u petlji koja ispisuje brojeve od `1` do `100`.

```
for (let i = 1; i <= 100; i++) {
  if (i === 15) {
    break; // Prekida petlju kada je i = 15, dakle neće se ispisati brojevi od 15 do 100
  }
  console.log(i); // ispisuje brojeve od 1 do 14
}
```

Kako bismo preskočili trenutnu iteraciju petlje, koristimo ključnu riječ `continue`. Kada se `continue` naredba izvrši, trenutna iteracija petlje se prekida i izvršavanje se nastavlja s idućom iteracijom petlje. Na primjer, želimo ispisati sve brojeve od `1` do `100` osim brojeva koji su djeljivi s 3.

```
for (let i = 1; i <= 100; i++) {
  if (i % 3 === 0) {
    continue; // Preskače trenutnu iteraciju petlje kada je i djeljiv s 3
  }
  console.log(i); // ispisuje sve brojeve od 1 do 100 osim brojeva koji su djeljivi s 3
}
```

`break` i `continue` naredbe možemo koristiti kod svih vrsta petlji - `for`, `while` i `do-while`.

`break` naredbu koristimo i unutar `switch` selekcija kako bi prekinuli njeno izvršavanje nakon ulaska u određeni `case` blok, međutim `continue` naredbu ne koristimo.

3.3.4 Petlje nad nizom znakova (eng. *String*)

Do sad smo koristili petlje za iteriranje kroz brojeve, ali možemo koristiti petlje i za iteriranje kroz nizove znakova. Na primjer, možemo ispisati svaki znak u nizu znakova. Kako bismo to postigli, koristimo `for` petlju i svojstvo `length` niza znakova koje nam govori koliko znakova niz sadrži. Kao i u C jezicima, indeksi znakova u nizu znakova počinju od `0` i idu do `length - 1`, a dohvaćamo ih koristeći operator `[]`.

```
let grad = "Pula";
for (let i = 0; i < grad.length; i++) {
  console.log(grad[i]); // ispisuje svaki znak u nizu znakova -> P, u, l, a
}
```

Idemo upotrijebiti svo znanje o petljama, selekcijama i funkcijama kako bismo napisali funkciju koja će zbrojiti ponavljanja određenog znaka u nizu znakova. Funkcija `brojPonavljanjaZnaka()` prima dva argumenta - niz znakova `niz` i znak `znak`. Funkcija vraća broj ponavljanja znaka `znak` u nizu znakova `niz`.

```
function brojPonavljanjaZnaka(niz, znak) {
  let brojac = 0;
  for (let i = 0; i < niz.length; i++) {
    if (niz[i] === znak) {
      // Provjerava je li trenutni znak u nizu znakova jednak znaku koji tražimo
      brojac++;
    }
  }
  return brojac;
}
console.log(brojPonavljanjaZnaka("Pula", "a")); // ispisuje broj ponavljanja znaka `a` u nizu znakova Pula -> 1
console.log(brojPonavljanjaZnaka("JavaScript", "a")); // ispisuje broj ponavljanja znaka `a` u nizu znakova JavaScript -> 2
console.log(brojPonavljanjaZnaka("JavaScript", "z")); // ispisuje broj ponavljanja znaka `z` u nizu znakova JavaScript -> 0
```

3.3.5 Ugniježđene petlje

Ugniježđene petlje koriste se kada želimo iterirati kroz više dimenzija podataka. Na primjer, kada želimo ispisati sve parove `(i, j)` brojeva u rasponu od `1` do `3`:

```
for (let i = 1; i <= 3; i++) {
  for (let j = 1; j <= 3; j++) {
    console.log(i, j); // ispisuje sve kombinacije parova brojeva od 1 do 3 -> 1 1, 1 2, 1 3, 2 1, 2 2, 2 3, 3 1, 3 2, 3 3
  }
}
```

Kombinirati i ugnijezditi i različite vrste petlji, na primjer, `for` i `while` petlje:

```
let i = 1;
while (i <= 3) {
  for (let j = 1; j <= 3; j++) {
    console.log(i, j); // ispisuje sve kombinacije parova brojeva od 1 do 3 -> 1 1, 1 2, 1 3, 2 1, 2 2, 2 3, 3 1, 3 2, 3 3
  }
  i++;
}
```

`break` i `continue` naredbe u ugniježđenim petljama ponašaju se kao i kod jednostavnih petlji - prekidaju petlju ili preskaču trenutnu iteraciju petlje u kojoj se izvršavaju.

```
for (let i = 1; i <= 3; i++) {
  for (let j = 1; j <= 3; j++) {
    if (i === 2 && j === 2) {
      continue; // Preskače iteraciju gdje je i = 2 i j = 2
    } else {
      console.log(i, j); // ispisuje sve kombinacije parova brojeva od 1 do 3 osim 2 2 ->
1 1, 1 2, 1 3, 2 1, 2 3, 3 1, 3 2, 3 3
    }
  }
}
```

Primjer 4 - Ispis tablice množenja

Primjenjujući ugniježđene petlje možemo jednostavno ispisati tablicu množenja. U ovom primjeru implementirat ćemo funkciju za ispis tablice množenja za brojeve od `1` do `10`. Funkcija će ispisati sve kombinacije brojeva od `1` do `10` i njihovih umnožaka.

Prvo definirajmo funkciju `tablicaMnozenja()` i unutar nje `for` petlju koja prolazi kroz brojeve od `1` do `10`.

```
function tablicaMnozenja() {
  for (let i = 1; i <= 10; i++) {
    console.log(i);
  }
}
tablicaMnozenja();
```

Dalje, želimo svaki broj `i` pomnožiti s brojevima od `1` do `10`. To ćemo jednostavno postići ugniježđenom `for` petljom.

```
function tablicaMnozenja() {
  for (let i = 1; i <= 10; i++) {
    for (let j = 1; j <= 10; j++) {
      console.log(i, j, i * j); // ispisuje sve kombinacije brojeva od 1 do 10 i njihove
umnožke
    }
  }
}
tablicaMnozenja();
```

Kako bismo dobili tablicu, možemo dodati i formatiranje ispisa. Na primjer, možemo koristiti tabulator `\t` kako bi razdvojili brojeve za veličinu jednog taba.
U varijablu `red` spremamo sve umnoške brojeva `i` i `j` od `1` do `10`, odvajamo ih tabulatorom, a zatim ispisujemo napunjeni `red` u vanjskoj petlji.

Rješenje:

```
function tablicaMnozenja() {
  for (let i = 1; i <= 10; i++) {
    let red = "";
    for (let j = 1; j <= 10; j++) {
      red += i * j + "\t";
    }
    console.log(red);
  }
}
tablicaMnozenja();
```

Vježba 5

Napišite program koji će ispisati sve brojeve od `1` do `100`. Za brojeve koji su djeljivi s 3 umjesto broja ispišite `Fizz`, za brojeve koji su djeljivi s 5 ispišite `Buzz`, dok za brojeve koji su djeljivi i sa 3 i sa 5 ispišite `FizzBuzz`. Ne ispisujte svaku vrijednost koristeći `console.log()`, već pohranjujte vrijednosti u varijablu `output` i na kraju ispišite niz koristeći `console.log(output)`. Nakon svake vrijednosti dodajte zarez i razmak (`, `), osim nakon posljednje vrijednosti, nakon nje dodajte `i kraj!`.

EduCoder šifra: `fizz_buzz`

Rezultat:

```
1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11,      script.js:18
Fizz, 13, 14, FizzBuzz, 16, 17, Fizz, 19, Buzz, Fizz, 22, 23, Fizz,
Buzz, 26, Fizz, 28, 29, FizzBuzz, 31, 32, Fizz, 34, Buzz, Fizz, 37,
38, Fizz, Buzz, 41, Fizz, 43, 44, FizzBuzz, 46, 47, Fizz, 49, Buzz,
Fizz, 52, 53, Fizz, Buzz, 56, Fizz, 58, 59, FizzBuzz, 61, 62, Fizz,
64, Buzz, Fizz, 67, 68, Fizz, Buzz, 71, Fizz, 73, 74, FizzBuzz, 76,
77, Fizz, 79, Buzz, Fizz, 82, 83, Fizz, Buzz, 86, Fizz, 88, 89,
FizzBuzz, 91, 92, Fizz, 94, Buzz, Fizz, 97, 98, Fizz, Buzz i kraj!
```

Vježba 6

Napišite funkciju koja prima jedan argument `godina` i provjerava je li godina prijestupna ili nije. Prema Gregorijanskom kalendaru, godina je prijestupna ako:

- je dijeljiva s 4, ali nije dijeljiva s 100
- ako je dijeljiva s 100, mora biti i s 400
  Na primjer, godine `1700`, `1800` i `1900` nisu prijestupne, ali godina `2000` jest.

EduCoder šifra: `svake_prijestupne`

Rezultat:

```
console.log(leapyear(2016)); // true
console.log(leapyear(2000)); // true
console.log(leapyear(1700)); // false
console.log(leapyear(1800)); // false
console.log(leapyear(100)); // false
```

3.4 Rekurzija (eng. *Recursion*)

Rekurzija je proces kada funkcija poziva samu sebe. Rekurzivne funkcije koriste se kada je problem koji rješavamo moguće podijeliti na manje probleme iste vrste. Rekurzivne funkcije koriste se za rješavanje problema koji se mogu svesti na manje probleme iste vrste, kao što su problemi vezani uz matematičke operacije, npr. faktorijela, Fibonaccijev niz, Tower of Hanoi, itd.

Rekurzivne funkcije imaju dvije komponente: bazni slučaj i rekurzivni slučaj. Bazni slučaj je uvjet koji prekida rekurziju, a rekurzivni slučaj je uvjet koji poziva samu funkciju za rješavanje manjeg problema iste vrste.

Primjer rekurzivne funkcije za izračun faktorijele broja `n`. Faktorijel broja `n` označava se s `n!` i definira je kao umnožak svih pozitivnih cijelih brojeva manjih ili jednakih `n`. Na primjer, faktorijel broja `5` označava se kao `5!` i iznosi `5 * 4 * 3 * 2 * 1 = 120`.

```
function faktorijel(n) {
  if (n === 0) {
    return 1; // Bazni slučaj, prekida rekurziju
  } else {
    return n * faktorijel(n - 1); // Rekurzivni slučaj, poziva samu funkciju za rješavanje manjeg problema iste vrste
  }
}
```

Rekurzija koristi stog memorije za pohranu svakog poziva funkcije. Ako se rekurzija ne prekine, može doći do prekoračenja stoga memorije i do rušenja aplikacije. Zato je važno osigurati da rekurzija ima bazni slučaj koji prekida rekurziju.

Kako izgleda poziv funkcije `faktorijel(5)`?

1. Početni poziv - `faktorijel(5)`

- bazni slučaj: `5` nije `0`, stoga se izvršava rekurzivni slučaj
- rekurzivni slučaj: `5 * faktorijel(4)`, dakle mora se izračunati `faktorijel(4)`

2. Poziv - `faktorijel(4)`

- bazni slučaj: `4` nije `0`, stoga se izvršava rekurzivni slučaj
- rekurzivni slučaj: `4 * faktorijel(3)`, dakle mora se izračunati `faktorijel(3)`

3. Poziv - `faktorijel(3)`

- bazni slučaj: `3` nije `0`, stoga se izvršava rekurzivni slučaj
- rekurzivni slučaj: `3 * faktorijel(2)`, dakle mora se izračunati `faktorijel(2)`

4. Poziv - `faktorijel(2)`

- bazni slučaj: `2` nije `0`, stoga se izvršava rekurzivni slučaj
- rekurzivni slučaj: `2 * faktorijel(1)`, dakle mora se izračunati `faktorijel(1)`

5. Poziv - `faktorijel(1)`

- bazni slučaj: `1` nije `0`, stoga se izvršava rekurzivni slučaj

- rekurzivni slučaj: `1 * faktorijel(0)`, dakle mora se izračunati `faktorijel(0)`

6. Poziv - `faktorijel(0)`

- bazni slučaj: `0` je `0`, stoga se rekurzija prekida i vraća se `1`

7. Vraćanje vrijednosti - `faktorijel(0)` vraća `1` u poziv `faktorijel(1)`
8. Vraćanje vrijednosti - `faktorijel(1)` vraća `1 * 1 = 1` u poziv `faktorijel(2)`
9. Vraćanje vrijednosti - `faktorijel(2)` vraća `2 * 1 = 2` u poziv `faktorijel(3)`
10. Vraćanje vrijednosti - `faktorijel(3)` vraća `3 * 2 = 6` u poziv `faktorijel(4)`
11. Vraćanje vrijednosti - `faktorijel(4)` vraća `4 * 6 = 24` u poziv `faktorijel(5)`
12. Vraćanje vrijednosti - `faktorijel(5)` vraća `5 * 24 = 120`

Dakle konačni rezultat poziva `faktorijel(5)` je `120`.

Rekurzija nije uvijek najbolje rješenje za rješavanje problema. Rekurzivne funkcije mogu biti teže za razumjeti i održavati, a mogu dovesti i do prekoračenja stoga memorije. U praksi, rekurzija se koristi kada je problem koji rješavamo mogu se svesti na manje probleme iste vrste, a rekurzivno rješenje je jednostavnije i čitljivije od iterativnog rješenja.

3.5 Primjer 5 - Validacija forme

Recimo da imamo web formu koja sadrži polja za unos imena, prezimena, e-maila i lozinke. Želimo provjeriti jesu li sva polja ispravno popunjena prije nego što se forma pošalje na server. Kako možemo to postići koristeći JavaScript?

Prvo, deklarirat ćemo 4 varijable koje će pohraniti vrijednosti unesene u polja forme.

```
let ime;
let prezime;
let email;
let lozinka;
```

Dalje, deklarirat ćemo funkciju `validirajFormu(ime, prezime, email, lozinka)` koja će provjeriti jesu li sva polja ispravno popunjena. U praksi, funkcija će se pozivati kada korisnik klikne na gumb za slanje forme. Ako funkcija vrati `true`, podaci u formi će se poslati na server, a ako vrati `false`, podaci neće biti poslani i korisnika će se na neki način obavijestiti.

Idemo prvo dodati provjere da su sva polja popunjena. Ako nisu, funkcija vraća `false` i obavještava korisnika koristeći `alert()` funkciju.

```
function validirajFormu(ime, prezime, email, lozinka) {
  if (ime === "" || prezime === "" || email === "" || lozinka === "") {
    alert("Molimo da popunite sva polja forme!");
    return false;
  }
  return true;
}
```

Dodajmo našim varijablama proizvoljne vrijednosti:

```
ime = "Sanja";
prezime = "Sanjić";
email = "sanjasanjic@gmail.com";
lozinka = "123456";

console.log(validirajFormu(ime, prezime, email, lozinka)); // ispisuje true
```

Sada ćemo dodati provjeru da lozinka mora sadržavati barem `6` znakova.

```
function validirajFormu(ime, prezime, email, lozinka) {
  if (ime === "" || prezime === "" || email === "" || lozinka === "") {
    alert("Molimo da popunite sva polja forme!");
    return false;
  }
  if (lozinka.length < 6) {
    alert("Lozinka mora biti dugačka barem 6 znakova!");
    return false;
  }
  return true;
}
```

U redu, što ako korisnik za ime i prezime unese brojeve? Dodajmo provjeru da ime i prezime sadrže samo slova.
Unutar naše funkcije `validirajFormu()` dodajmo pomoćnu funkciju `containsNumber()` koja će provjeriti sadrži li niz znakova brojeve.

Funkcija `containsNumber()` radi na način da prolazi kroz svaki znak niza i provjerava je li znak broj. Ako je, vraća `true`, inače vraća `false`.

> JavaScript pokušava pretvoriti znakove u brojeve kada koristimo operator `>=` i `<=`, stoga to možemo iskoristiti za usporedbu znakova s brojevima.
> Ako znak (`string`) predstavlja broj (0 - 9), JavaScript ga uspješno pretvara u broj (`number`) i provodi aritmetičku provjeru.

```
function validirajFormu(ime, prezime, email, lozinka) {
  if (ime === "" || prezime === "" || email === "" || lozinka === "") {
    alert("Molimo da popunite sva polja forme!");
    return false;
  }

  if (lozinka.length < 6) {
    alert("Lozinka mora biti dugačka barem 6 znakova!");
    return false;
  }

  function containsNumber(str) {
    for (let i = 0; i < str.length; i++) {
      // Provjerava je li znak broj (0-9)
```

```
      if (str[i] >= 0 && str[i] <= 9) {
        return true;
      }
    }
    return false;
  }

  if (containsNumber(ime) || containsNumber(prezime)) {
    alert("Ime i prezime ne smiju sadržavati brojeve!");
    return false;
  }

  return true;
}
```

Sada ako pokušamo pozvati funkciju `validirajFormu(ime, prezime, email, lozinka)` s proizvoljnim argumentima, funkcija će provjeriti jesu li sva polja ispravno popunjena, je li lozinka dugačka barem `6` znakova i sadržavaju li ime i prezime brojeve.

```
ime = "Sanja";
prezime = "Sanji3";
email = "sanjasanjic@gmail.com";
lozinka = "123456";

console.log(validirajFormu(ime, prezime, email, lozinka)); // false
```

U Javascriptu, znakovi (uključujući i brojeve i slova) se kodiraju koristeći Unicode skup znakova. U ASCII i Unicode skupovima znakova, znakovi se prikazuju numeričkim vrijednostima. Primjerice, u **ASCII** skupu, slovo `a` kodira se brojem `97`, a slovo `z` brojem `122`. Brojevi se kodiraju brojevima od `48` do `57`. Dok u **Unicode** skupu, znak `0` kodira se brojem `0030`, a znak `9` brojem `0039`.
Imajući to na umu, možemo dodati novu provjeru za `ime` i `prezime`. Funkciju koja provjerava sadrže li ime i prezime samo niz znakova `[a - z]`.

Na ovaj način ne uzimamo u obzir hrvatska slova: `č`, `ć`, `š`, `đ`, `ž`, `lj`, `nj`, `dž`.

```
function containsOnlyLetters(str) {
  for (let i = 0; i < str.length; i++) {
    let c = str[i];
    // Leksikografska usporedba znakova
    if (!(c >= "a" && c <= "z") && !(c >= "A" && c <= "Z")) {
      return false;
    }
  }
  return true;
}
```

> U ovom slučaju, JavaScript će znakove pretvoriti u brojeve prema **Unicode** skupu znakova. Dano rješenje radi zato što znamo da su slova `[a - z]` kodirana brojevima od `97` do `122` i od `65` do `90` u ASCII skupu znakova. Drugim riječima, pohranjena su sekvencijalno!

Zapamtite:

> - **containsNumber(str)** : Ovdje su nam operandi u selekciji **znak** i **broj**. JavaScript će u ovom slučaju nastojati pretvoriti znak u broj i provesti usporedbu.
> - **containsOnlyLetters(str)** : Ovdje su nam operandi u selekciji **znak** i **znak**. JavaScript će u ovom slučaju usporedti znakove leksikografski, tj. po redoslijedu u ASCII skupu znakova.

Možemo još dodati provjeru je li e-mail ispravno formatiran. Na primjer, e-mail mora sadržavati znak `@` i barem jednu točku nakon znaka `@`.
Radi pojednostavljenja, nećemo provjeravati sadrži li e-mail nedozvoljene znakove ili više znakova `@`.

Naša konačna funkcija `validirajFormu()` izgleda ovako:

```
function validirajFormu(ime, prezime, email, lozinka) {
  if (ime === "" || prezime === "" || email === "" || lozinka === "") {
    alert("Molimo da popunite sva polja forme!");
    return false;
  }

  if (lozinka.length < 6) {
    alert("Lozinka mora biti dugačka barem 6 znakova!");
    return false;
  }

  function containsNumber(str) {
    for (let i = 0; i < str.length; i++) {
      // Provjerava je li znak broj (0-9)
      if (str[i] >= "0" && str[i] <= "9") {
        return true;
      }
    }
    return false;
  }

  function containsOnlyLetters(str) {
    for (let i = 0; i < str.length; i++) {
      let c = str[i];
      // Leksikografska usporedba znakova
      if (!(c >= "a" && c <= "z") && !(c >= "A" && c <= "Z")) {
        return false;
      }
    }
    return true;
  }

  function checkEmail(email) {
    let atFound = false;
    let dotFound = false;
    for (let i = 0; i < email.length; i++) {
      if (email[i] === "@") {
        atFound = true;
      }
```

```
      if (atFound && email[i] === ".") {
        dotFound = true;
      }
    }
    return atFound && dotFound;
  }

  if (containsNumber(ime) || containsNumber(prezime)) {
    alert("Ime i prezime ne smiju sadržavati brojeve!");
    return false;
  }

  if (!containsOnlyLetters(ime) || !containsOnlyLetters(prezime)) {
    alert("Ime i prezime smiju sadržavati samo slova a-z!");
    return false;
  }

  if (!checkEmail(email)) {
    alert("Email mora sadržavati @ i najmanje jednu točku (.) nakon @!");
    return false;
  }

  return true;
}
```

Samostalni zadatak za vježbu 3

Napomena: Ne predaje se i ne boduje se. Zadatak možete i ne morate rješavati u EduCoder aplikaciji.

EduCoder šifra: `ReversePrimeLongest`

1. Napišite funkciju `reverseString` koja prima znakovni niz (string) kao argument i vraća obrnuti string. Na primjer, ako je ulaz `"hello"`, funkcija treba vratiti `"olleh"`. Funkcija mora vratiti `"Not a string!"` ako je ulazni argument različitog tipa od stringa. Funkciju testirajte s argumentima `"hello"`, `"JavaScript"` i `123`.
2. Napišite funkciju `prost_broj` koja prima broj kao argument i vraća `true` ako je broj prost, odnosno `false` ako nije. Broj je prost ako je djeljiv samo s 1 i samim sobom. Funkciju pozovite s argumentima 7, 10 i 13.
3. Nadogradite prethodni zadatak na način da ćete ispisati sve proste brojeve od 1 do 100. Funkciju `prost_broj` pozivajte unutar petlje. Ispis mora izgledati ovako: `"Prosti brojevi od 1 do 100 su: 2, 3, 5, 7, itd."`
4. Napišite funkciju `pronadi_najduzu_rijec()` koja prima rečenicu kao argument i vraća najdužu riječ u rečenici. Rečenicu morate razložiti **koristeći petlju**, **bez pomoćnih funkcija/metoda**!
   - Ako se funkciji proslijedi tip podatka koji nije string, funkcija vraća `"Nije rečenica!"`.
   - Ako je rečenica prazna, funkcija vraća `"Rečenica je prazna!"`.
   - Ako se rečenica sastoji od samo jedne riječi, funkcija vraća tu riječ.
   - Ako se rečenica sastoji od više različitih najdužih riječi, funkcija vraća prvu riječ koja je pronađena.