**JS prototipovi (20)**

JavaScript je objektno-orijentisani programski jezik koji ne koristi klasični koncept klasa i nasleđivanja, već je u osnovi jezik baziran na prototipovima. Svaki objekat u sebi sadrži referensu na odgovarajući prototip, koji u sebi sadrži referensu na svoj prototip, itd. Na ovaj način dobijamo lanac prototipova koji se završava objektom čija je referensa na prototip jednaka null. Obično je to Object.prototype. Na primer, svaki instanciran niz će kao svoj prototip imati ugrađeni Array.prototype:

let x = new Array(1,2,3,4,5);

console.log(Object.getPrototypeOf(niz));

Prototipu možemo pristupiti koristeći metodu Object.getPtototypeOf(imeInstance). Prototip od Array.prototype je zatim Object.prototype:

let x = new Array(1,2,3,4,5);

let prototip = Object.getPrototypeOf(niz);

let protoPrototip = Object.getPrototypeOf(prototip);

console.log(protoPrototip);

Prototip od Object.prototype je jednak null. Dakle, počev od niza x, dobijamo sledeći prototipski lanac:

x → Array.prototype → Object.prototype → null

Slično, svaka funkcija će kao svoj prototip imati Function.prototype, kom je prototip Object.prototype:

nekaFunkcija → Function.prototype → Object.prototype → null

Ukoliko želimo da pristupimo određenom atributu nekog objekta, taj objekat prvo traži takav atribut kod sebe, a ako ga ne nađe, traži ga kod svog prototipa, pa zatim kod prototipa svog prototipa, itd. Ako imamo datu funkciju funk, možemo instancirati objekat ob koristeći operator new primenjen na datu funkciju, pri čemu ta funkcija igra ulogu konstruktora:

function funk() {

this.a = 11;

this.b = 17;

}

let ob = new funk();

console.log(ob);

console.log(Object.getPrototypeOf(ob));

console.log(Object.getPrototypeOf(Object.getPrototypeOf(ob)));

console.log(Object.getPrototypeOf(funk));

Prototip objekta ob je funk.prototype, čiji je prototip Object.prototype. Prototip same funkcije funk je Function.prototype, kao što smo već utvrdili. Generalno nije pametna ideja menjati sadržaj ugrađenih prototipova, ali možemo bez problema da menjamo sadržaj prototipa funk.prototype. Na primer:

function funk(aa, bb) {

this.a = aa;

this.b = bb;

}

funk.prototype.c = 997;

let ob1 = new funk(15, 17);

let ob2 = new funk(23, 2);

console.log(ob1.c);

console.log(ob2.c);

Na ovaj način smo napravili da objekti ob1 i ob2 imaju zajednički atribut c, zato što on pripada njihovom zajedničkom prototipu. Međutim, ukoliko bismo pokušali da upišemo u taj atribut objekta ob1:

ob1.c = 16;

console.log(ob1.c);

objekat bi napravio novi atribut c koji se nalazi direktno u samom objektu, a ne u prototipu, i njemu bi dodelio datu vrednost. Dakle, ako hoćemo da menjamo vrednost atributa c tako da to vide svi objekti, mora da se pristupa prototipu funk.prototype. Ovo nam daje mogućnost da simuliramo princip instanciranja više objekata iste klase, poznat u objektno-orijentisanim jezicima kao što su C++, Java i C#:

function Student(name, surname, gpa) {

this.ime = name;

this.prezime = surname;

this.prosek = gpa;

}

Student.prototype.imeFaksa = "Elfak";

Student.prototype.getIme = function() {

return this.ime;

}

Student.prototype.getPrezime = function() {

return this.prezime;

}

Student.prototype.getProsek = function() {

return this.prosek;

}

Student.prototype.getFull = function() {

return this.ime + " " + this.prezime + " " + this.prosek + " " + this.imeFaksa;

}

var s = new Student("Marko", "Markovic", 8.7);

var t = new Student("Pera", "Peric", 9.2);

console.log(s.getFull());

console.log(t.getFull());

Student.prototype.imeFaksa = "ETF";

console.log(s.getFull());

console.log(t.getFull());

U ovom slučaju pravimo dva objekta iste „klase“, Student. Postoje 3 atributa objekata: ime, prezime i prosek, kao i 1 atribut klase (statički atribut): imeFaksa. Sve metode su pridružene prototipu zato što izvršavaju isti kod, i nema potrebe da se troši dodatni prostor u memoriji i da se pamti metoda za svaki objekat nezavisno. Kada se poziva metoda, specifikator this se ne odnosi na objekat u kom se nalazi metoda (prototip), već na objekat koji je poziva.

U datom primeri, objekti imaju javne atribute. Ako bismo hteli da atributi postanu privatni, to se može postići na sledeći način:

var Student = (function() {

var imeFaksa = "Elfak";

return function(name, surname, gpa) {

var ime = name;

var prezime = surname;

var prosek = gpa;

this.getIme = function() {

return ime;

}

this.getPrezime = function() {

return prezime;

}

this.getProsek = function() {

return prosek;

}

this.getFull = function() {

return ime + " " + prezime + " " + prosek + " " + imeFaksa;

}

this.setFaks = function(faks) {

imeFaksa = faks;

}

}

})();

var s = new Student("Marko", "Markovic", 8.7);

var t = new Student("Pera", "Peric", 9.2);

console.log(s.getFull());

console.log(t.getFull());

s.setFaks("ETF");

console.log(s.getFull());

console.log(t.getFull());

Koristi se princip closure kako bi se omogućilo pamćenje privatnog statičkog atributa imeFaksa, kom se ne može pristupiti spolja, a može iz javne metode setFaks. Na sličan način se za svako izvršenje konstruktorske funkcije u njenom kontekstu pamte privatne promenljive ime, prezime, prosek, kojim takođe ne može pristupiti spolja (opet se closure koristi). Ispis će biti:

Marko Markovic 8.7 Elfak

Pera Peric 9.2 Elfak

Marko Markovic 8.7 ETF

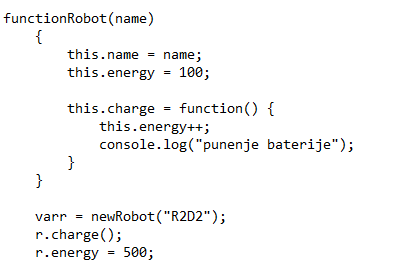
Pera Peric 9.2 ETF

Kada preko jednog objekta promenimo vrednost imeFaksa, ta promena se vidi i u svim ostalim objektima. Primetimo da se u ovom primeru metode ne mogu realizovati kao metode prototipa, već je potrebno da se direktno dodeljuju samim objektima, zato što koriste njihove privatne atribute.

## **22. JS ES5 klase**

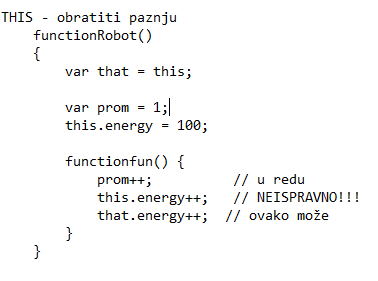
U JS-u postoje određeni mehanizmi objektno orijentisane paradigme ali prototipskog tipa poput one u prvim oo jezicima. U ES5 još uvek nije uvedena ključna reč *class,* te nam u ove svrhe pomaže CLOSURE mehanizam. Postoje 3 načina definisanja klasa.

1. Klasu definišemo praktično definisanjem njene konstruktorske funkcije:



Imenima metoda dodeljujemo anonimne funkcije, a unutar konstruktora dodeljujemo atributima (propertijima) vrednost preko ključne reči this. Pri tome treba voditi računa da ne smemo da koristimo ključnu reč this unutar lokalnih funkcija kako ono tehnički ne ukazuje na objekat koji kreiramo. Takve funkcije se mogu samo dodeliti metodima klase i pozivati se tek nakon inicijalizacije celog objekta, odnosno završetka cele konsktruktorske funkcije.

Dodavanje metoda u klasu se može izvršiti na još dva načina:



functionApple(type){

this.type = type;

this.color = "red";

this.getInfo = getAppleInfo;

}

// anti-pattern! keep reading...

functiongetAppleInfo(){

returnthis.color + '' + this.type + ' apple';

}

Međutim, prethodni način dodavanja funkcija nije preporučljiv jer se definicije funkcija navode u globalnom namespace-u. Takođe može doći do konflikta u dodeljivanju imena ako se dodaje globalna funkcija sa istim imenom, ili se uključuje biblioteka koja sadrži funkciju sa istim imenom.

Korišćenjem prethodna 2 načina metode se prave svaki put kada se pravi novi objekat. Treći način dodavanja metoda u klasu je dodavanje metode prottipu konstuktorske klase.

Function Apple(type){

this.type = type;

this.color = "red";

}

Apple.prototype.getInfo = function(){

returnthis.color + '' + this.type + ' apple';

};

1. Korišćenjemliteralaobjekata

Literali su skraćeni način da se definiše objekat ili niz u js. Kako bi se napravio prazan objekat dovoljno je korisiti:

var o = {};

Korišćenjem literala moguće je automatsko pravljenje klase:

varapple = {

type: "macintosh",

color: "red",

getInfo: function(){

returnthis.color + '' + this.type + ' apple';

}

}

U ovomslučaju nije moguće praviti instancu klase jer ona već postoji, pa se odmah kreće sa njenim korišćenjem:

apple.color = "reddish";

alert(apple.getInfo());

Ovakviobjekti se nazivaju *singleton.*

1. Pravljenje singleton korišćenjem funkcije

Treći način je kombinacija prva dva, koristi se funkcija kako bi se napravio singleton:

Var apple = newfunction(){

this.type = "macintosh";

this.color = "red";

this.getInfo = function(){

returnthis.color + '' + this.type + ' apple';

};

}

Nasleđivanje je moguće pomoću prototipova.

// Shape - superclass

// x,y: location of shape's bounding rectangle

**function** Shape(x, y) {

**this**.x = x;

**this**.y = y;

}

// Superclass method

Shape.prototype.move = **function**(x, y) {

**this**.x += x;

**this**.y += y;

}

// Circle - subclass

**function** Circle(x, y, r) {

// Call constructor of superclass to initialize superclass-derived members.

Shape.call(**this**, x, y);

// Initialize subclass's own members

**this**.r = r;

}

// Circle derives from Shape

Circle.prototype = Object.create(Shape.prototype);

Circle.prototype.constructor = Circle;

// Subclass methods. Add them after Circle.prototype is created with

// Object.create

Circle.prototype.area = **function**() {

**returnthis**.r \* 2 \* Math.PI;

}

## 23. JS ES6

**BONUS: ES5 vs ES6**- uveden je lamba izraz =>  
(x) => { return x\*x; } ili x => x\*x;  
- u ES6 se koristi let, u ES5 var  
- uvedene su konstante  
const PI = 3,141593;  
- uvedena kljucna rec class za kreiranje klasa  
class Student { … }  
- uvedeno nasledjivanje klasa isto kao kod OOP jezika  
class Student extends Osoba {…} //super() poziv konstruktora nasledjene klase  
- simboli === I !== koji se koriste za uporedjivanje ne samo vrednosti nego I tipova  
5 I “5” nije isto  
…..i jos mnogo toga.

ECMAScript 2015 donosi puno novina koje olakšavaju programiranje i doprinose preglednosti koda. Većina njih predstavlja sintaksički šećer, odnosno ne uvodi nove paradigme već olakšava postizanje istog efekta na jednostavniji način. Neke od njih su:

* **Ključne reči class i constructor**

Class Rectangle{

constructor(height, width){

this.height = height;

this.width = width;

}

// Getter

Get area(){

Return this.calcArea();

}

// Method

calcArea(){

return this.height \*this.width;

}

}

Klase se sada mogu deklarisati mnogo prirodnije kao i njeni metodi. Uvedena je kljucna reč constructor za deklarisanje konstruktora, a mogu se jednostavno definisati getteri i metodi.

* **Extend**

Moguće je nasleđivanje klasa korišćenjem reči extend

Class Animal{

constructor(name){

this.name = name;

}

speak(){

console.log(this.name +' makes a noise.');

}

}

Class Dog extends Animal{

speak(){

console.log(this.name +' barks.');

}

}

var d =new Dog('Mitzie');

d.speak();// Mitzie barks.

* **Ključna reč *const***

Ima blokovski opseg važenja, poput ključne reči let i omogućava defnisanje konstante.

// NOTE: Constants can be declared with uppercase or lowercase, but a common

// convention is to use all-uppercase letters.

// define MY\_FAV as a constant and give it the value 7

const MY\_FAV =7;

// this will throw an error - Uncaught TypeError: Assignment to constant variable.

MY\_FAV =20;

// MY\_FAV is 7

console.log('my favorite number is: '+ MY\_FAV);

// trying to redeclare a constant throws an error - Uncaught SyntaxError: Identifier 'MY\_FAV' has already been declared

const MY\_FAV =20;

// the name MY\_FAV is reserved for constant above, so this will fail too

var MY\_FAV =20;

// this throws an error too

let MY\_FAV =20;

* **Ključna reč *let***

Uvedena je ključna reč let sa blokovskim opsegom važenja koja sprečava veoma rasprostranjene greške koje se čine pomoću var promenljive koja se može preklopiti u bilo kom ugnježdenom bloku.

functionvarTest(){

var x =1;

if(true){

var x =2;// same variable!

console.log(x);// 2

}

console.log(x);// 2

}

Function letTest(){

let x =1;

if(true){

let x =2;// different variable

console.log(x);// 2

}

console.log(x);// 1

}

Takođe, *let* ne definiše property nad globalnom promenljivom:

var x ='global';

let y ='global';

console.log(this.x);// "global"

console.log(this.y);// undefined

* **Arrow funkcije**

Anonimne funkcije se sada mogu pisati u skraćenom, lambda obliku koji čini kod mnogo preglednijim.

var materials =[

'Hydrogen',

'Helium',

'Lithium',

'Beryllium'

];

materials.map(function(material){

return material.length;

});// [8, 6, 7, 9]

materials.map((material)=>{

return material.length;

});// [8, 6, 7, 9]

materials.map(({length})=> length);// [8, 6, 7, 9]

Arrow funkcije rešavaju problem sa this promenljivom tako što ako se definišu unutar klase, promenljiva this zaista ukazuje na objekat te klase:

functionPerson(){

this.age =0;

setInterval(()=>{

this.age++;// |this| properly refers to the person object

},1000);

}

var p =newPerson();

Sintaksa:

### Basic Syntax

(param1, param2, …, paramN) => { statements }

(param1, param2, …, paramN) => expression

// equivalent to: => { return expression; }

// Parentheses are optional when there's only one parameter name:

(singleParam) => { statements }

singleParam => { statements }

// The parameter list for a function with no parameters should be written with a pair of parentheses.

() => { statements }

* **Default vrednosti parametara**

Moguće je definisati default vrednosti parametara

Function multiply(a, b =1){

return a \* b;

}

multiply(5,2);// 10

multiply(5,1);// 5

multiply(5);// 5

* **Object.assign()**

Ova metoda kopira propertije jednog objekta u propertije drugog, koristi get i set metode.

Moguće je spajanje (merge) objekata, kao i kloniranje.

var o1 ={ a:1};

var o2 ={ b:2};

var o3 ={ c:3};

var obj = Object.assign(o1, o2, o3);

console.log(obj);// { a: 1, b: 2, c: 3 }

console.log(o1);// { a: 1, b: 2, c: 3 }, target object itself is changed.

Kloniranje:

var obj ={ a:1};

var copy = Object.assign({}, obj);

console.log(copy);// { a: 1 }

* **Interpolacija stringova...**

Moguće je formirati višelinijske stringove i vrednosti promenljivih unutra stringa na jednostavniji način:

var a =5;

var b =10;

console.log(`Fifteen is ${a + b} and

not ${2\* a + b}.`);

// "Fifteen is 15 and

// not 20."

**TypeScript (opšte, poređenje sa JS) (25)**

Tajpskript je besplatan programski jezik otvorenog koda napravljen od strane Majkrosofta. Tajpskript je sintaksički nadskup Javaskripta (ECMAScript 6), pa je svaki validan Javaskript program istovremeno validan Tajpskript program. Za Tajpskript se često kaže da je to Javaskript sa tipovima što i jeste istina jer je glavna novina koju Tajpskript donosi statička tipizacija. Tajpskript je jezik čiji se izvorni kod kompajlira u Javaskript (ECMAScript 5). To znači da praktično i dalje imamo dinamičku tipizaciju na nivou Javaskript koda dobijenog kompilacijom tj. da se tip svake promenljive određuje u vreme izvršenja, s tim da ovde imamo jedan sloj iznad koji nam pruža prividnu statičku tipizaciju tako što će sva neuklapanja tipova (najčešće kada se funkciji prosleđuju argumenti neodgovarajućeg tipa ili prilikom dodele) biti prijavljenja kao greška (greška u vremenu prevođenja).

Kao sledeći primer dat je jednostavan Javaskript program:

|  |
| --- |
| function greeter(person) {  return "Hello, " + person;  }  let user = "Lesbian User";  document.body.innerHTML = greeter(user); |

Ovo je, naravno, istovremeno validan Tajpskript program. Kada se radi o Tajpskriptu, ovom programu možemo dodati tipove.

|  |  |
| --- | --- |
| Tajpskript program | Rezultat prevođenja u Javaskript |
| function greeter(person: string) {  return "Hello, " + person;  }  let user1 = "Gay User";  let user2 = [0, 1, 2];  alert(greeter(user1));  alert(greeter(user2)); | function greeter(person) {  return "Hello, " + person;  }  var user1 = "Gay User";  var user2 = [0, 1, 2];  alert(greeter(user1));  alert(greeter(user2)); |

U poslednjem primeru, kod drugog poziva biće prijavljena greška još u vreme prevođenja. Ako bismo poslednji primer ispravili u:

|  |  |
| --- | --- |
| Tajpskript program | Rezultat prevođenja u Javaskript |
| function greeter(person: string) {  return "Hello, " + person;  }  let user1 = "Gay User";  let user2 = "Bisexual User";  alert(greeter(user1));  alert(greeter(user2)); | function greeter(person) {  return "Hello, " + person;  }  var user1 = "Gay User";  var user2 = "Bisexual User";  alert(greeter(user1));  alert(greeter(user2)); |

Moguće je naglašavati tipove prilikom deklaracije promenljivih:

|  |
| --- |
| Tajpskript program |
| let fullName: string = `Bisexual Bob`;  let age: number = 37;  let sentence: string = `Hello, my name is ${ fullName }.I'll be ${ age + 1 } years old next month.`;  let list: number[] = [1, 2, 3];  let words: string[] = ['Lesbian','Gay','Bisexual','Transgeder']; |
| Rezultat prevođenja u Javaskript |
| var fullName = "Bisexual Bob";  var age = 37;  var sentence = "Hello, my name is " + fullName + ".I'll be " + (age + 1) + " years old next month.";  var list = [1, 2, 3];  var words = ['Lesbian', 'Gay', 'Bisexual', 'Transgeder']; |

**TypeScript klase i nasleđivanje (sa primerom) (26)**

Tajpskript podržava klase nalik ECMAScript 6 klasama sa opcionim navođenjem informacija o tipovima. Bitno je napomenuti da je ovo bilo moguće pre nego što je ECMAScript 6 u potpunosti završen. Najbolje je poledati jednostavan primer:

|  |  |
| --- | --- |
| Tajpskript program | Rezultat prevođenja u Javaskript |
| class Person {  private name: string;  private age: number;  private salary: number;  constructor(name: string, age: number, salary: number) {  this.name = name;  this.age = age;  this.salary = salary;  }  } | var Person = /\*\* @class \*/ (function () {  function Person(name, age, salary) {  this.name = name;  this.age = age;  this.salary = salary;  }  return Person;  }()); |

Pogledaćemo još jedan primer:

|  |
| --- |
| Tajpskript program |
| class Greeter {  greeting: string;  constructor(message: string) {  this.greeting = message;  }  greet() {  return "Hello, " + this.greeting;  }  }  let greeter = new Greeter("world");  let button = document.createElement('button');  button.textContent = "Say Hello";  button.onclick = function() {  alert(greeter.greet());  }  document.body.appendChild(button); |
| Rezultat prevođenja u Javaskript |
| var Greeter = /\*\* @class \*/ (function () {  function Greeter(message) {  this.greeting = message;  }  Greeter.prototype.greet = function () {  return "Hello, " + this.greeting;  };  return Greeter;  }());  var greeter = new Greeter("world");  var button = document.createElement('button');  button.textContent = "Say Hello";  button.onclick = function () {  alert(greeter.greet());  };  document.body.appendChild(button); |

Postoji mogućnost da ispred imena argumenata konstruktora stavimo modifikator public. Dejstvo toga je da ce biti napravljeni atributi sa tim imenom sa dodeljenom prosleđenom vrednošću.

|  |
| --- |
| Tajpskript program |
| class Student {  private fullName: string;  constructor(public firstName: string, public middleInitial: string, public lastName: string) {  this.fullName = firstName + "" + middleInitial + "" + lastName;  }  }  let user = new Student("Jane", "M.", "User"); |
| Rezultat prevođenja u Javaskript |
| var Student = /\*\* @class \*/ (function () {  function Student(firstName, middleInitial, lastName) {  this.firstName = firstName;  this.middleInitial = middleInitial;  this.lastName = lastName;  this.fullName = firstName + "" + middleInitial + "" + lastName;  }  return Student;  }());  var user = new Student("Jane", "M.", "User"); |

Tajpskript omogućava i nasleđivanje. Prvo ćemo pogledati jednostavniji primer a onda jedan koji je malo složeniji. Oba primera ne zahtevaju dodatno objašnjavanje s obzirom da je nasleđivanje nalik C# ili Java nasleđivanju. Kod koji se dobija prevođenjem neće biti naveden jer je malo veće dužine (prevodi se u ECMAScript 5), može se dobiti na https://www.typescriptlang.org/play.

|  |
| --- |
| Prvi primer |
| class Animal {  static kingdom = "Animalia";  move(distanceInMeters: number = 0) {  console.log(`Animal moved ${distanceInMeters}m.`);  }  }  class Dog extends Animal {  bark() {  console.log('Woof! Woof!');  }  }  const dog = new Dog();  dog.bark();  dog.move(10);  dog.bark(); |
| Drugi primer |
| class Animal {  name: string;  constructor(theName: string) { this.name = theName; }  move(distanceInMeters: number = 0) {  console.log(`${this.name} moved ${distanceInMeters}m.`);  }  }  class Snake extends Animal {  constructor(name: string) { super(name); }  move(distanceInMeters = 5) {  console.log("Slithering...");  super.move(distanceInMeters);  }  }  class Horse extends Animal {  constructor(name: string) { super(name); }  move(distanceInMeters = 45) {  console.log("Galloping...");  super.move(distanceInMeters);  }  }  let sam = new Snake("Sammy the Python");  let tom: Animal = new Horse("Tommy the Palomino");  sam.move();  tom.move(34); |

**TypeScript šabloni (sa primerom) (28)**

Tajpskript poseduje generički mehanizam koji je sintaksno gotovo istovetan C# generičkom mehanizmu. Generičke klase, funkcije i interfejsi u Tajpskriptu se prevode u običan ECMAScript 5 kod koji nema tipove podataka (sve informacije vezane za tipove se određuju u vreme izvršenja). Pogledaćemo kako funkcionišu generičke funkcije i generičke klase.

Jednostavna generička funkcija bila bi:

|  |  |
| --- | --- |
| Tajpskript program | Rezultat prevođenja u Javaskript |
| function loggingIdentity<T>(arg: T): T {  console.log(arg);  return arg;  } | function loggingIdentity(arg) {  console.log(arg);  return arg;  } |

Ova funkcija prima argument proizvoljnog tipa, loguje prosleđeni argument u konzolu i vraća isti taj prosleđeni argument. Vidimo da je sintaksa ista kao kod C# generičkog mehanizma s tim što se tipovi argumenata i tip povratne vrednosti drugačije navodi. Primećujemo da je kod dobijen prevođenjem negenerički. Zaključujemo da je smisao Tajpskript generičkog mehanizma da samo pruži strogu formalizaciju u izvornom kodu (Javaskript je jezik sa dinamičkim tipovima pa mu generički mehanizam nije neophodan da ostvari neke stvari koje generički mehanizmi omogućavaju jezicima sa statičkim tipovima).

Pogledaćemo jedan jednostavan primer sa klasama koji ne zahteva dodatna objašnjenja:

|  |
| --- |
| Tajpskript program |
| class Greeter<T> {  greeting: T;  constructor(public message: T) {}  greet() {  return this.greeting;  }  }  let greeter1 = new Greeter<string>("Hello, world");  console.log(greeter1.greet());  let greeter2 = new Greeter<number>(987123);  console.log(greeter2.greet()); |
| Rezultat prevođenja u Javaskript |
| var Greeter = /\*\* @class \*/ (function () {  function Greeter(message) {  this.message = message;  }  Greeter.prototype.greet = function () {  return this.greeting;  };  return Greeter;  }());  var greeter1 = new Greeter("Hello, world");  console.log(greeter1.greet());  var greeter2 = new Greeter(987123);  console.log(greeter2.greet()); |

Generički mehanizam u Tajpskriptu podržava i dodavanje ograničenja. Ograničenja se navode posle ključne reči extends (koja je navodi posle identifikatora tipa) a sve unutar <> zagrada. Ograničenje može biti definisano uz pomoć klase ili uz pomoć interfejsa. Prvo ćemo pogledati primer sa ograničenjem preko zadate klase. Kada se posle ključne reči extends navede identifikator neke klase, to znači da generički tip može biti ili ta klasa ili neka klasa izvedena iz te klase. Primer pojašnjava stvari:

|  |
| --- |
| Tajpskript program |
| class Greeter<T extends String> {  greeting: T;  constructor(public message: T) {}  greet() {  return this.greeting;  }  }  class MyString extends String{  id : number;  private name: number;  constructor(str: String) {  super(str);  this.id = 0;  }  }  let greeter1 = new Greeter<string>("Hello, world");  console.log(greeter1.greet());  let greeter2 = new Greeter<MyString>(new MyString("Hello, LGBT"));  console.log(greeter2.greet()); |

Ograničenja mogu biti zadata i preko interfejsa. Šta ako na primer želimo da postignemo sledeće:

|  |
| --- |
| Tajpskript program |
| class Greeter<T extends String> {  greeting: T;  constructor(public message: T) {}  greet() {  this.greeting.print();  }  } |

Ovde će biti prijavljena greška koja glasi da tip T nema metodu print. Rešenje je da napravimo jednostavan interfejs IPrintable i da ga dodamo kao ograničenje za tip T. Kada imamo više ograničenja ona se nadovezuju znakom &.

|  |
| --- |
| Tajpskript program |
| interface IPrintable {  print() : void;  }  class Greeter<T extends String & IPrintable> {  greeting: T;  constructor(public message: T) {}  greet() {  this.greeting.print();  }  }  class MyString extends String{  constructor(str: String) {  super(str);  }  print() : void {  console.log(this);  }  }  let greeter = new Greeter<MyString>(new MyString("Hello, LGBT"));  greeter.greet(); |

**Single Page Application (48)**

Single Page Application (SPA) je web aplikacija kod koje se celokupna interakcija sa korisnikom odvija na jednoj stranici. Takva stranica se učita jednom i dinamički ažurira tokom korišćenja. Jedna od najpopularnijih tehnologija za razvoj SPA su JavaScript frejmvorci AngularJS i React.

Pri interakciji sa tradicionalnim web aplikacijama server odgovara zahtevu klijenta slanjem **nove stranice** koja se ucitava na strani klijenta. Kod SPA aplikacija serveru se upućuje AJAX zahtev i onodgovara **potrebnim podacima**, najčešće u JSON formatu. Posledice ovakvog pristupa su:

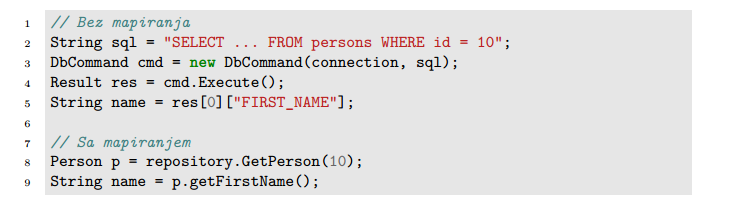
* Razdvajanje prezentacionog dela od logike i podataka, što iziskuje bolje projektovanje aplikacije, postiže se stabilnost i aplikacija se lakše proširuje. Projektovanje klijenta je u potpunosti nezavisno od projektovanja servera. Klijent samo treba da zna koji zahtev da pošalje, prikazivanje tih podataka radi samostalno.

Komplikovaniji klijentski deo. Server ne daje nikakve informacije o prikazu podataka koje šalje. Klijent dobija podatke od servera i ceo user interface realizuje sam. Iz tog razloga je potrebno uložiti više rada u client-side i frontend i u tu svrhu se koristi JavaScript.

**Pitanje 43** ORM alati.

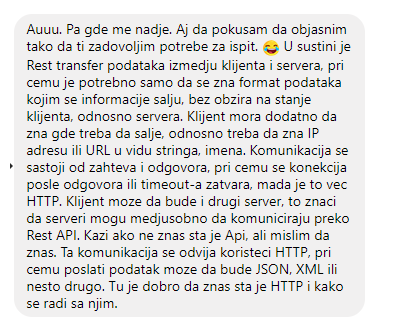
*Odgovor.* ORM (**Object-Relational Mapping**) je programska tehnika za pretvaranje podataka u objektno-orijentisanim programskim jezicima između međusobno nekompatiblnih struktura. Na ovaj način se kreira “virtuelna objektna baza podataka” koja se može koristiti iz programskog jezika. U objekto-orijentisanom programiranju, upravljanje podacima bazira se na objektno-orijentisanim objektima koji su skoro uvek neskalarne veličine. Na primer, stavka u adresaru reprezentuje jednu osobu, koja ima nula ili više brojeva telefona, i nula ili više adresa. Ovo se u objekto-orijentisanom modelu može implementirati definisanjem objekta Person koji ima atribute koji drže podatke o njemu: ime osobe, lista brojeva telefona i lista adresa. Lista brojeva telefona bi u sebi sadržala objekte PhoneNumber, i tako dalje. Stavka u adresaru se tretira kao jedan objekat od strane programskog jezika, i može mu se pristupiti jednom promenljivom (na primer, pokazivačem koji pokazuje na objekat). Nad ovim objektom se mogu primenjivati različite metode, npr. vraćanje podrazumevanog broja telefona, itd.

Međutim, mnogi popularni alati za rad sa bazama podataka, kao što je SQL DBMS, mogu da čuvaju i manipulišu samo skalarnim veličinama kao što su celi brojevi i stringovi, organizovani u tabele. Programer mora ili da konvertuje vrednosti objekata u grupe sa jednostavnijim vrednostima za čuvanje u bazi (i da ih ponovo konvertuje pri upisu nazad), ili da koristi samo jednostavne skalarne vrednosti u programu. Objektno-orijentisano mapiranje implementira prvi prilaz. Srce problema leži u prevođenju logičke reprezentacije objekata u atomičnu formu koju je moguće skladištiti u bazi podataka, održavajući pritom svojstva objekata i njihove veze tako da se mogu ponovo učitati kao objekiti kada se za tim javi potreba. Ako se implementira ova funkcionalnost za pribavljanje i skladištenje, za objekte se kaže da su **perzistentni**.

  
**Veb-servisi  
Pitanje 44** Veb-servisi.

*Odgovor.* W3C definiše veb-servis kao “softverski sistem dizajniran tako da podrži interoperabilnu interakciju između mašina preko mreže”. Za razmenu podatka se koristi XML ili JSON. Veb-servis obično pruža objektno-orijentisani interfejs serveru baze podataka, koji se koristi od strane drugog veb-servera ili mobilne aplikacije, koji nudi korisnički interfejs krajnjem korisniku. Poznati primeri su Google Maps API i EBay Shopping API. Veb-servisi mogu biti SOAP (*Simple Object Access Protocol*, tipičan primer service1.getPerson() ) ili REST (*Representational State Trasnfer*, tipičan primer http://example.com/persons ).

**Pitanje 47** RESTful servisi.

  
*Odgovor.* REST (**Representational State Transfer**) je arhitekturni stil koji se sastoji od koordinišućeg skupa komponenti, konektora i elemenata u distribuiranom hipermedija sistemu, gde je fokus na ulogama komponenti i određenom skupu interakcija među elementima, a ne na implementacionim detaljima. REST je arhitekturni

obrazac koji se koristi na WWW-u. Predstavio ga je Roj Filding 2000. godine. Ima za svrhu da poboljša sledeće:

*•* **performanse** – interakcija komponenti može biti dominantan factor u korisnikovom doživljaju o performansama,

*•* **prilagodljivost** (skalabilnost) – treba da može da podrži veliki broj komponenti i interakcij¯a među njima,

*•* **jednostavnost** interfejsa,

*•* **promenljivost** (modifikabilnost) komponenti kako bi se udovoljilo zahtevima za promenama, čak i dok je

aplikacija pokrenuta,

*•* **vidljivost** (vizibilitet) komunikacije među komponentama od strane servisnih agenata,

*•* **prenosivost** (portabilnost) komponenti prenošenjem programskog koda sa podacima, i

*•* **pouzdanost**, tj. otpornost na otkaz na nivou celog sistema u prisustvu otkaza njegovih pojedinih komponenti, konektora ili podataka.

U REST arhitekturi, REST server samo pruža pristup resursima, a REST klijent pristupa i predstavlja resurse. Svaki resurs je identifikovan preko URI-ja ili preko globalnog ID-ja. Na primer, URI /UserService/users prosleđene preko GET može da vrati listu svih korisnika, dok navođenje konkretnog parametra (/UserService/users/1) može

da vrati konkretnog korisnika (sa ID-jem 1). REST koristi razne reprezentacije za predstavljanje resurs¯a, kao što su tekst, JSON ili XML. Danas je JSON najpopularniji format. Za veb-aplikacije koje su u skladu sa gorenavedenim osobinama se kaže da su RESTful.

Pitanje 45 SOAP servisi.

*Odgovor.* SOAP (**Simple Object Access Protocol**) je protokol računarskih mreža za razmenu struktuiranih informacija u implementaciji vebservisa. Koristi XML Information Set kao format poruka. Za transmisiju i pregovore se oslanja na protokole apliaktivnog nivoa, najčešće HTTP ili SMTP. SOAP može biti osnovni sloj za stek protokola veb-servisa, jer može da pruži osnovni frejmvork za razmenu poruka veb-servisima. Baziran je na XML-u I sastoji se od tri dela:  
*•* **koverta**, koja definiše strukture poruke i način na koji se one obrađuju,  
*•* **skup pravila** za šifrovanje za iskazivanje instanci tipova podatakadefinisanih od strane aplikacije i  
*•* **konvenciju** za reprezentaciju poziva procedura i odgovora.  
Tri glavne karakteristike SOAP-a su sledeće:  
*•* **proširljivost**,  
*•* **neutralnost** (može koristiti bilo koji protokol za transport, kao što suHTTP, SMTP, TCP, UDP, itd) i  
*•* **nezavisnost** (dozvoljava bilo koji programski model).  
Na primer, aplikacija može da pošalje SOAP zahtev serveru na kome su omogućeni veb-servisi – kao što je baza podataka sa cenama nekretnina – sa parametrima pretrage. Server zatim vraća SOAP odgovor (document formatiran kao XML koji sadrži rezultate), npr. cenu, lokaciju i osobine. Kako generisani podaci dolaze u standardizovanom formatu koji mašine mogu parsirati, aplikacija koja je pružila zahtev može direktno integrisati odgovor.



**Prednosti** SOAP-ova neutralnost ga ekspliticno čini pogodnim za korišćenje sa bilo kojim transportnim protokolom. Implementacije često koriste HTTP, ali se mogu koristiti i drugi (SMTP). SOAP, kada se kombinuje sa HTTP post/odgovor razmenama, lako kanališe kroz postojeće fajervolove I proksije, i stoga ne zahteva modifikaciju široko rasprostranjenih komunikacionih infrastruktura koje već postoje za obradu HTTP razmena.  
**Mane** Kada se koriste standardne implementacije i podrazumevane kombinacije SOAP/HTTP, XML infoset je serijalizovan kao XML. Ako je neophodno ugraditi binarne objekte u XML, radi boljih performansi se koristi  
MTOM (*Message Transmission Optimization Mechanism*).  
**Pitanje 46** RPC.  
*Odgovor.* RPC (**Remote Procedure Call**) je opis situacije u kojoj kompjuterski program primorava izvršenje procedure u drugom adresnom prosotru (često na drugom računaru u mreži), pri čemu je kod napisan kao da se radi o klasičnom (lokalnom) pozivu procedure, odnosno pri čemu programmer nije u dužnosti da eksplicitno kodira detalje takve udaljene konekcije. Drugim rečima, programer piše isti kod bilo da se procedura poziva lokalno u odnosu na program koji se izvršava, ili ne. Ovo je primer klijent-server interakcije (pozivalac je klijent, izvršilac je server), i obično se implementira sistemom za razmenu poruka “zahtev-odgovor”. Pandam RPC-u u objektnoorijentisanom programiranju je RMI (Remote Method Invocation).

RPC-ovi su vrsta inter-procesne komunikacije (IPC), u kojoj različiti procesi imaju različite adresne prostore: ako su na istoj mašini, imaju različite virtuelne adresne prostore i pored toga što je fizički adresni prosotor isti; ako su na različitm hostovima, fizički adresni prostor je različit.  
RPC započinje klijent, koji šalje poruku poznatom udaljenom serveru u kojoj nalaže izvršenje specificirane procedure koristeći priložene parametre. Udaljeni server šalje odgovor klijentu, i aplikacija nastavlja svoj proces. Dok server obrađuje poziv, klijent je blokiran (čeka dok mu server ne odgovori), osim ako je zahtev bio poslat asinhrono, kao što je XHTTP poziv. Važna razlika između udaljenih poziva procedura i lokalnih poziva jeste to što RPC može da ne uspe zbog nepredviđenih problema sa mrežom. Sem toga, u opštem slučaju, pozivalac mora da se nosi sa takvim padovima neznajući da li je udaljena procedura uopšte pozvana. Idempotentne procedure (procedure koje nemaju dodatne efekte ako se pozovu više od jedanput, npr. Metoda DELETE) ne predstavljaju problem, ali ostale procedure ostaju problem.  
Sekvenca događaja tokom RPC-a je sledeća.  
1. Klijent poziva klijentski stab (*stub*). To je deo koda koji se koristi za konvertovanje parametara koji se prenose tokom RPC-a. Na primer, to može biti prevođenje celih brojeva iz adresiranja glave u adresiranje  
repa. Ovaj poziv je klasičan lokalni poziv procedure, i parametri se potavljaju na stek na uobičajen način.  
2. Klijentska stab procedura pakuje parametre u poruku i inicira sistemski poziv kako bi se poruka poslala. Pakovanje parametara se naziva maršaling (*marshalling*).  
3. Klijentov lokalni operativni sistem šalje poruku od klijentske mašine ka serverskoj mašini.  
4. Lokalni operativni sistem na serveru mašine prenosi dobijene pakete na serverski stab.  
5. Serverski stab raspakuje parametre iz poruke (tzv. demaršaling).  
6. Konačno, serverski stab poziva proceduru. Odgovor prati iste korake, samo u obrnutom redosledu.

Pitanje 4 Arhitektura veb-aplikacija.

MVC je softverski arihtekturni obrazac koji se cesto koristi za implementaciju korisnickih interfejsa; stoga predstavlja  
popularan izbor za kreiranje arihtekture veb-aplikacija. Najopstije, MVC deli aplikacionu logiku na tri odvojena dela,  
promovisuci modularnost, jednostavnu medjusobnu saradnju I mogucnost ponocvnog koriscenja.  
1)Model -definise sta aplikacija treba da sadrzi. Ako se stanje ovih podatak promeni, onda model uglavnom  
treba da obavesti pogled (kako bi se prikazala nastala promena), a ponekad kontroler(za slucaj da je potrebna  
drugacija logika da vi se kontrolisao pogled koji se azurira(koje torte sadrzi lista)  
2)Pogled-definise kako se prikazuju podaci aplikacije I prima podatke koje treba da prikaze od modela(Primer  
kako se prezentuje lista tortu koju treba naruciti)

3)Kontroler-sadrzi logiku koja azurira model i/ili pogleed kao odgovor na promene koje korisnik unosi  
podatke . (nasa aplikacija sadrzi polja za unos I kako da obradi unete torte ili izabrane torte. Ove akcije zahtevaju da  
se model azurira, zato unos se salje kontroleru, koji zatim manipulise modelom na odgovarajuci nacin. Kontroler  
moze takodje da iskoristi stanje da promeni neke vrednosti bez koriscenja modela, jednostavno samo primeniti  
odredjenu akciju.  
Kod veb aplikacija model podataka je najcesce sadrzan u nekoj bazi podataka (MySQL, ili resenje na klijentskoj stani  
IndexedDB). Kod koji kontrolise aplikaciju uglavnom je napisan koriscenjem HTML I JavaScrip-a, dok je dinamicki  
interferjs( pogled) deginisan HTML-om, CSS-om I mozda JavaScriptom. U ranim razvojima veba, MVC se nalazio kao  
implementiran na serverskoj stani, pri cemu je klijent zahtevao azuriranje podatak koriscenjem formi ili putem  
linkova, a primao je azurirane podatke kao stranice koje su se prikazivale u brauzeru. Medjutim, u modernom dobu  
sve vise logike se prebacuje na stranu kolenta zhvaljujuci napretku skladistenje podataka na klijentu. Pored  
toga,objekti tipa XMLHttpRequesr dopustaju samo azuriranje dela stranice. Populari web frejmovrci(frejmvork  
predstavlja konceptualnu strukturu koja ima za cil opsluzivanje I ucestvovanje u razvoju neceg korisnog. U  
kompjuterskom svetu, softverski frejmvork obezbedjuje funkcionalnosti koje se mogu menjati od strane programera  
ili web dizajnera kako bi se dobio softver ili web dizajn oanakav kako je predhodno zamisljem), kao sto su Angular JS,  
Ember.js I Backbone implementiraju MVC arhitekturu, mada svaki na drugaciju nacin. Kada projektujemo i  
dizajniramo poslovne informacione sisteme primenjujemo višeslojnu arhitekturu.U praksi se može identifikovati  
nekoliko stilova arhitekture:  
• Klijent/Server : predstavlja razdvajanje sistema u dve aplikacije: klijentska, koja se nalazi kod korisnika i  
baza podataka sa aplikativnom logikom  
• Arthitektura zasnovana na komponentama: sistem čine funkcionalne ili logič•ke komponente  
koje obezbeđuju interfejs za spoljnu komunikaciju  
• Domenski vođen dizajn: Objektno orjentisan pristup koji je fokusiran na modelovanje sistema kreiranjem  
klasa identifikovanih u poslovnom domenu koji se modelira  
• N-slojeva / 3 sloja: troslojna arhitektura je jedna od najčešće korišćenih u praksi na  
osnovu dosadašnjeg iskustva. Troslojnu arhitekturu čine: prezentacioni, sloj poslovne logike i sloj podataka.  
• Servisno orjentisana arhitektura (SOA): odnosi se na aplikacije koje obezbeđuju i koriste funkcionalnosi  
kao servise koristeći ugovore i poruke  
Arhitektura softverskog sistema najčešće predstavljaju kombinaciju gore navedenih stilova.Iako se svi podjednako  
koriste u praksi, detaljnije možemo obraditi troslojnu arhitekturu.  
Troslojnu arhitekturu čine tri sloja:  
 1. Prezentacioni – ovaj sloj obezbeđuje prikaz podataka krajnjem korisniku koristeći neku od raspoloživih  
tehnologijaÂ za korisnič•ki interfejs. U sluč•aju da pravite desktop aplikacije to može biti Windows Forms ili WPF  
dok recimo ako pravite web aplikacije to ASP .NET.  
 2. Sloj poslovne logike – ovaj sloj implementira poslovnu logiku aplikacije. Poslovnu logiku čine poslovni procesi i  
poslovne komponente. Ovde se takođe najčešće implentiraju i poslovna pravila dobijena u procesu analize.  
 3. Sloj podataka – većina poslovnih aplikacija koristi relacione baze podataka za skladiš¡tenje podataka.  
Neke od prednosti ovakve arhitekture su:  
• Skalabilnost ( sposobnost programa da uspeÅ¡no obradi porast korisnika, npr. u web aplikacijama, skalabilnost bi  
bila sposobnost aplikacije da uspeš¡no funkcioniše kod povećanog broja poseta. Pored programskih zahteva da se  
poveća skalabilnost, ne treba zaboraviti i hardverske komponente koje pokreću sistem jer i one mogu uticati na  
ovaj kvalitativni atribut)  
• Fleksibilnost ( sposobnost programa da se lako prilagodi novim zahtevima korisnika)  
• Dostupnost ( verovatnoća da sistem ispravno funkcioniše bilo kad, osim kada je u pitanju održavanje ) – ova  
prednost se ogleda u tome što je arhitektura sistema modularna)  
• Održavanje  
Troslojna arhitektura je pogodna za sledeće scenarije:  
• intranet aplikacije gde su serveri postavljeni unutar privatne mreže  
• internet aplikacija kada je dozvoljeno da poslovna logika bude dostupna na javno pristupač•nim web ili  
aplikativnim serverima  
-veb aplikacije struktuirane su u tri glavna sloja(takozvano troslojnu arhitekturu) :  
 1)Prezentacioni sloj-sloj prikaza informacija korisniku purem pretrazivaca (Google chrom, MS internet  
explorer)

2)Aplikacioni sloj-sloj koji upravlja aktivnostima koje aplikacije treba izvrsavati (tj.obradom ili  
funkcionisanjem). (dinamicke veb tehnologije, ASP, PHP)

3)sloj Podataka-sloj koji upravla slanjem podata u bazu I vracanje podata iz baze… (SQL , MySqly)  
**Pitanje 5** Opšte o HTML-u. Struktura stranice.

*Odgovor.* HTML (**HyperText Markup Language**) je standardni jezik za obeležavanje koji se koristi za kreiranje veb-stranica. Uz CSS i JavaScript, predstavlja temeljnu tehnologiju za izgradnju veb-stranica, kao i korisnič-  
kih interfejsa za mobilne i veb-aplikacije. Veb-čitači čitaju HTML fajlove i renderuju ih, što ga čini **jezikom za obeležavanje** (*markup*), a ne programskim jezikom. HTML semantički opisuje strukturu veb-stranice, a pre  
pojave CSS-a sadržao je i opis izgleda stranice. Gradivni elementi HTML stranice su **HTML elementi**. Pomoću njih je moguće kreiranje strukturnih dokumenata obeležavanjem naslov¯a, paragraf¯a, listi, linkova, citat¯a, itd. HTML elementi su omeđeni tagovima, koji se pišu u uglasnim zagradama. Na primer, tekst *primer* je omeđen parom tagova (tag i anti-tag) na sledeći način: <**p**>primer</**p**> . Postoje i tagovi koji nemaju svoj  
anti-tag, već zatvaraju sami sebe, kao što je <**img** /> (slika) ili <**input** /> (polje za unos).  
Primer minamalne HTML stranice dat je u nastavku.



Tekst između <**html**> i </**html**> opisuje veb-stranicu, a tekst između <**body**> i </**body**> opisuje vidljivi deo stranice. Zaglavlje HTML dokumenta nalazi se između para tagova <**head**> i </**head**> i u njemu se između ostalog može naći naslov stranice (u ovom primeru <**title**>Naslov</**title**> ).

**Pitanje 6** HTML5, semantički elementi.

*Odgovor.* Semantički elementi jasno opisuju svoje značenje, kako veb-čitaču, tako i programeru. Pored bolje čitljivosti koda, važni su zbog pristupačnosti (npr. za audio-čitače veb-stranica za slepe) i za SEO (optimizacija za vebpretraživače). Dva najčešće korišćena **nesemantička** taga su <**div**> i <**span**> .

*HTML Document Division Element* ( <**div**> ) je **generički kontejner** za sadržaj, koji nema posebno značenje i ne predstavlja ništa. Može da se koristi za grupisanje elemenata radi stilizovanja (koristeći atribute id i class ), ili  
za elemente koji dele zajedničke atribute, kao što je lang . Treba se koristiti samo kada nijedan drugi semantički element ne odgovara svrsi.  
*HTML Document Span Element* ( <**span**> ) je **generički kontejer** koji se prostire u okviru linije teksta (inlajn element), i ne predstavlja ništa. Koristi se kao i <**div**> , s tim što je <**div**> element na nivou bloka, dok je <**span**> na nivou linije.  
Prema MDN-u, semantički elementi se dele na osnovne elemente, metapodatke, elemente za izdvajanje sadržaja, elemente koji sadrže tekst, inlajn elemente, elemente za slike i druge multimedijalne sadržaje, elemente za ugrađeni (*embedd*) sadržaj, elemente za skripte, elemente za demarkaciju, elemente za tabelarne podatke, elemente za forme, interaktivne elemente I veb-komponente.  
**Osnovni elementi** Jedini osnovni element koji svaka stranica mora da ima da bi se mogla smatrati HTML stranicom jeste <**html**> . Ovaj element predstavlja koren HTML dokumenta. Svi elementi moraju biti njegovi potomci.  
**Metapodaci o dokumentu** Element <**base**> specificira osnovni URL koji treba da koriste svi relativni URL-ovi iz dokumenta.

Element <**head**> daje opšte informacije o dokumentu, uključujući naslov I linkove do skirpti i stajlšitova.  
Element <**link**> specificira vezu između trenutnog dokumenta i eksternog izvora; koristi se najčešće za povezivanje sa stajlšitovima.  
Element <**style**> koristi se za specifikaciju stilova koje treba primeniti na trenutni dokument. Podrazumevano se radi o CSS jeziku.  
Element <**title**> govori o naslovu stranice koja će biti prikazana u veb-čitaču.  
Element <**meta**> koristi se za podatke koji se ne mogu svrstati u gorenavedene metapodatke.  
**Izdvajanje sadržaja** Ovi elementi organizuju dokument u logičke celine. Pomoću njih se brzo kreira opšta kontura stranice.  
Element <**article**> predstavlja celinu koja može postojati sama za sebe I može se ponovo koristiti u drugom kontekstu. To može biti poruka na forumu, članak u časopisu, post na blogu, objekat, ili bilo koji drugi nezavistan sadržaj. Svaki <**article**> treba da bude identifikovan, obično dodavnjem elemenata za naslov (npr. <**h1**> ) kao deteta artikla.  
Element <**section**> predstavlja generičku sekciju dokumenta, npr. Tematsko grupisanje sadržaja, najčešće sa naslovom. Kao i <**article**> , svaki <**section**> treba da bude identifikovan.  
Elementi od <**h1**> do <**h6**> predstavljaju šest nivoa naslov¯a u dokumentu, pri čemu manji redni broj označava veću važnost. Ovaj element kratko opisuje temu sekcije koju predstavlja. Ove informacije agenti mogu da koriste da bi automatski napravili sadržaj. Svaki <**section**> ima svoju hijerarhiju naslov¯a, tako da sledeći isečak HTML stranice predstavlja hijerarhiju u kome je *Opet naslov* podnaslov naslova *Naslov*, iako su oba sa istim rednim brojem.



Element <**header**> predstavlja grupu sadržaja uvodnog ili navigacionog karaktera. Može sadržati nasove, ali i druge elemente koji predstavljaju npr. logo, formu za pretragu, itd.

Element <**footer**> predstavlja futer za najbližu roditeljsku sekciju (ili za ceo dokument, ako je najbliža sekcija predstavljena <**body**> tagom). Futer najčešće sadrži podatke o autoru sekcije, informacije o zaštiti autorskih prava i slično.  
Element <**nav**> koristi se za deo stranice koji sadrži linkove do drugih stranica ili do raznih delova iste stranice. Dakle, to je sekcija sa navigacionim linkovima.  
Element <**address**> dostavlja kontakt-inforamcije u vezi sa najbližim <**article**> roditeljem. Ako takav ne postoji, podaci se tiču celog dokumenta.  
**Tekst sadržaj** Ovi elementi se koriste radi organizacije blokova ili sekcija. Važni su zbog pristupačnosti i SEO jer identifikuju značaj sadržaja koji se u njima sadrži.  
Element <**ul**> predstavlja neuređenu listu stavki, odnosno kolekciju stavki koje nisu ni na koji način numerisani. Slično, <**ol**> predstavlja uređenu listu. Unutar ovih elemanta se mogu naći isključivo <**li**> elementi, i njihov sadržaj predstavlja stavke koje treba nabrojati. Brauzeri stavke najčešće renderuju jednu ispod druge, dodajući simbole ispred stavki: bulet u slučaju neuređene liste i redni broj u slučaju uređene liste. Prilagođavanje ovih osobina vrši se putem CSS-a, a ne atributima liste.  
Element <**main**> predstavlja glavni sadržaj elementa <**body**> u dokumentu ili aplikaciji. Sadržaj ovog elementa treba da bude od suštinske važnosti, i da ne uključuje delove koji se ponavljaju kroz stranice. Treba da bude jedinstven na stranici.  
Element <**p**> predstavlja paragraf u tekstu.  
Element <**pre**> predstavlja ranije formatirani tekst. Obično se renderuje korišćenjem neproporcionanog (monospejs) fonta i znaci beline se ne sažimaju kao kod ostalih elemenata.  
Element <**dl**> predstavlja listu opisa i omeđuje parove termina i njihovih opisa. Obično se koristi za implementaciju rečnika ili za prikaz metapodataka. Elementi <**dt**> i <**dd**> mogu da se nađu samo u listi opisa i predstavljaju, redom, termin i opis. Može ih biti više uzastopno (više termina ima isti opis ili jedan termin ima više opisa). Najpre se navodi termin (ili termini) pa opis (ili opisi). Brauzeri termine obično renderuju boldirano, a opise u novom redu, uvučeno.

Element <**figure**> pretstavlja sadržaj koji može postojati sam za sebe, najčešće uz natpis koji je predstavljen elementom <**figcaption**> . Mada je usko vezan za glavni tok, njegova pozicija ne zavisi od glavnog toka. Ovo je obično slika, ilustracija, dijagram, isečak koda ili šema koja je referencirana u tekstu, ali se može pomeriti na sledeću stranicu ili u apendiks, a da ne naruši glavni tok.  
Element <**hr**> po HTML5 specifikaciji predstavlja tematsku pauzu između elemanta na nivou paragrafa (na primer, promena scene u priči ili promena teme u članku). U ranijim verzijama HTML-a, ovaj element je predstavljao horizonalnu liniju bez semantičkog značenja. Mada ga veb-čitači i dalje najčešće predstavljaju na isti način, on sada ima semantičko značenje, a ne prezentaciono.

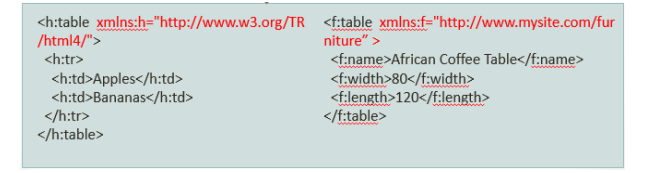
**Pitanje 7** CSS opste

CSS je jezik za formatiranje koji omogucava da se definise izgled elemenenata, prvenstveno veb-stranica mada se  
moze koristiti iza bilo koje XML dokumente. Uz HTML I JavaScript predstavlja temeljnu tehnologiju za izradu vebstranica. CSS je dizajniran kako bi pre svega odvojio sadrzaj dokumenata od prezentacije istog, pri cemo se pod  
prezentacijom najcesce podrazumeva raspored elemenata, boja, fontovi. Ovakva struktura omogucuje vecu  
flieksibilnost pri specifiaciji karakteristika prezentacije, jer omogucuje da vise stranica dele isti eksterni fajl (.css)  
cime se smanjuje ponavljanje isth podataka u strukturi samih dokumenata. Moguce je takodje putem css-a jednu  
stranicu prezentovati na razlicite nacine.

**Pitanje 8** CSS selektori.  
*Odgovor.* **Selektori** služe da se definišu na koje se delove dokumenta odnosi deklaracioni blok koji sledi. Selektori se mogu odnositi na:  
*•* sve elemente istog zadatog tipa (npr. naslov drugog nivoa <**h2**> ) –**selektor tipa** (npr. **h2** {color: red} );  
*•* elemente koji imaju zadati atribut ili zadatu vrednost atributa, ili su određenom odnosu za zadatom vrednošću atributa – **selektori atributa** (npr. [**checked**] , [**lang**="en"] , [**data-tag**~="post"] );  
*•* elemente koji imaju specificiran identifikator ili klasu:  
**–** identifikator koji je jedinstven u celom dokumentu – **selector identifikatora** ( **#logo** {color: red} )2,  
**–** identifikator koji može da se odnosi na više elemenata u dokumentu – **selektor klase** ( **.card** {color: red} )3;  
*•* sve elemente ( \* ) – **univerzalni selektor**. U tesnoj vezi sa selektorima su tzv. **kombinatori**.  
*•* Kombinator (spejs) bira čvorove koji su **naslednici** (ne obavezno direktni naslednici, tj. deca) prethodno specificiranog elementa. Na primer, **div span** se odnosi na sve elemente <**span**> koji se nalaze u  
elementu <**div**> .  
*•* Kombinator + selektira čvorove koji se nalaze odmah nakon specificiranog elementa (**susedna braća**). Na primer, **img** + **span** se odnosi na sve elemente <**span**> koji se nalaze odmah nakon elementa <**img**> (pa se tako mogu npr. selektirati opisi slika).  
*•* Kombinator ~ selektira čvorove koji se nalaze nakon (ne obavezno odmah nakon) specificiranog elementa (**opšta braća**). Na primer, **p** ~ **span** se odnosi na sve elemente <**span**> koji se nalaze nakon elementa <**p**> , a imaju zajedničkog roditelja.  
*•* Kombinator > selektira **decu**, odnosno **direktne potomke** prethodno specificiranog elementa. Na primer, **ul** > **li** se odnosi na sve elemente <**li**> koji su direktni potomci <**ul**> (pa se tako npr. mogu definisati stilovi za različite “dubine” ugnježdenih listi).  
Od selektora, koriste se još i pseudo-klase i pseudo-elementi. **Pseudo-klase** dozvoljavaju formatiranje u skladu sa informacijama koje nisu dostupne u stablu dokumenta; na primer, deklaracioni blok selektora :hover će se primeniti na element samo kad korisnik postavi kursor na element. Neke pseudoklase mogu imati i argumente, kao što su :nth-child() , pa tako pseudo-klasa :nth-child(**4**) selektira samo četvrti element u nizu. Selekciju je moguće načini i pomoću posebne promenljive *n*, :nth-child(**3n**) . Ovaj selektor selektira svaki treći element.  
**Pseudo-elementi**, za razliku od pseudo-klasa, ne opisuju posebna stanja, već omogućuju stilizovanje samo nekih delova dokumenta. Na primer, pseudoelement ::first-line gađa samo prvu liniju elementa specificiranog selektorom. Zvanično ih je podržano šest: ::after , ::before , ::first-letter ,  
::first-line , ::selection i ::backdrop .



**XML namespace**- XML namespace se koristi kako bi XML dokumentu obezbedili elemente sa jedinstvenim imenom. Ako se  
svakom dokumentu dodeli namespace, dvosmislenost izmedju elemenata sa istim imenom se može rešiti. XML  
namespace se deklariše korišćenjem rezervisanog XML atributa xmlns ili xmlns:prefiks. Primer:



**XML tehnologije**- XML (eXtensible Markup Language) je sličan HTML-u i služi za prenos podataka, a ne za njihovu prezentaciju  
čemu služi HTML. XML elementi nisu predefinisani, već ih korisnik sam definiše (XML Schema). Sintaxa XML-a:

