**CS102 - Skripta**

**GRUPA PITANJA 1**

**1.** Šta je JavaFX?

🡪 **JavaFX** predstavlja novu platformu za razvoj Java programa sa grafičkim korisničkim interfejsom (GUI).

**2.** Zašto je uvedena JavaFX?

🡪 **JavaFX** je uvedena kao zamena za AWT i Swing jer pojednostavljuje izradu GUI programa. Sadrži moderne GUI tehnologije koje omogućavaju razvoj desktop aplikacija. JavaFX podržava primenu 2D i 3D grafike i animacija, prikaz video i audio snimaka, a radi ili kao nezavisna desktop aplikacija ili korišćenjem čitača veba (Web browser). Takođe, omogućava podršku i za touch-screen uređaje.

**3.** Osnovna struktura JavaFX programa - opišite?

🡪 **Osnovna struktura:** Stage predstavlja osnovni objekat, tzv. pozornicu. To je prozor. U stage se smešta scena, koja sadrži čvorove (nodes). JavaFX je napravljena po ugledu na pozorište. Imamo glavnu pozornicu na kojoj se menjaju scene, a svaka scena sadrži glumce - čvorove.

**4.** Šta je okno? Šta je čvor? Šta su oblici u JavaFX?

🡪 **Čvor** (Node) je vizuelna komponenta oblik (Shape), prikaz slike (ImageView), UI kontrola ili okno (Pane).

🡪 **Okno** (pane) je kontejner klase koje automatski raspoređuje čvorove na određene lokacije i sa određenim veličinama.

🡪 **Oblik** (shape) može biti: tekst, krug, linija, elipsa, luk, pravougaonik, poligon, polilinija i dr.

🡪 **UI kontrola** može biti label, button, check box, radio button, text field, text area i dr.

**5.** Objasnite povezivanje osobina ciljnog i izvornog objekta?

🡪 JavaFX uvodi novi koncept **povezivanja svojstava** koji omogućava da ciljni objekat bude povezan sa izvornim objektom. Kada se promeni vrednost izvornog objekta, promeni se i vrednost ciljnog objekta. Ciljni objekat se naziva povezanim (binding) objektom, a oba objekta se nazivaju povezanim (bindable) objektima. Metod bind() povezuje ciljni objekat sa izvornim objektom.

**6.** Šta je definisano apstraktnom klasom Node?

🡪 Apstraktna klasa **Node** definiše mnoga svojstva i metode koji su zajednički za sve čvorove. Primera radi, JavaFX svojstva stila su slična kao CSS i definišu se sa prefiksom **-fx**. Višestruka svojstva nekog čvora se mogu podesiti zajedno, ali odvojena sa (;). Postoji i svojstvo **rotate** koje omogućava rotaciju čvora oko centra (specifičan ugao u stepenima).

**7.** Klase Paint i Color u JavaFX platformi?

🡪 JavaFX koristi apstraktnu klasu **Paint** za bojenje nekog čvora. Klasa **Color** je konkretna podklasa klase Paint koja sadrži metode za definisanje boja. Klasa Color koristi **RGBA** model, koji definiše neku boju kombinacijom crvene (**R**), zelene (**G**) i plave (**B**) boje i stepena prozirnosti (opacity) - alfa vrednost (**A**).

**8.** Objasnite funkcionalnosti klase Font?

🡪 Klasa **Font** služi za kreiranje fontova. Definiše ime fonta (vrste slova), njegovu postavku i veličinu. Objekat klase Font se kreira konstruktorima ili statičkim metodima. Statički metod **getFamilies()** daje listu raspoloživih fontova (npr. **Arial**, **Cursive**, **Times New** **Roman**..). Postavka fonta može biti: *kosim slovima* (*italic*), **izraženim slovima** (**bold**) i normalnim slovima (regular).

**9.** Objasnite funkcionalnosti klasa Image i ImageView?

🡪 Klasa **Image** predstavlja grafičku sliku i koristi se za unos slike sa neke datoteke ili sa URL-a. Klasa **ImageView** je čvor koji prikazuje neku sliku.

**10.** Šta je menadžer (okno) rasporeda? Koja okna poznajete i kako se ponašaju?

🡪 JavaFX obezbeđuje više okna koja automatski raspoređuju čvorove na željene lokacije sa željenom veličinom. **Okno** (**pane**) se koristi kao **platno** (**canvas**) za prikazivanje **oblika** (**shapes**). Tipovi okna:

* **Pane** 🡪 osnovna klasa za raspoređivanje čvorova
* **BorderPane** 🡪 okno koje postavlja čvorove na vrh, desno, dole, levo i na centar
* **GridPane** 🡪 okno koje postavlja čvorove u ćelije dvodimenzione rešetke (matrice)
* **HBox** 🡪 okno koje postavlja čvorove u jedan red (horizontalu)
* **VBox** 🡪 okno koje postavlja čvorove u jednu kolonu (vertikalu)
* **StackPane** 🡪 okno koje postavlja čvorove jedan iznad drugog u centru okvira
* **FlowPane** 🡪 okno koje postavlja čvorove red po red (po horizontali) ili kolona po kolona (po vertikali)

**11.** Opisati detaljno primenu okna FlowPane?

🡪 **FlowPane** prestavlja okno koje ređa čvorove po horizontali (red po red) ili po vertikali (kolona po kolona). Kada se popuni jedan red ili kolona, počinje ređanje u drugom redu ili koloni. Primenom dve konstante (**Orientation.HORIZONTAL** ili **Orientation.VERTICAL**), određuje se željeno ređanje, po horizontali ili vertikali.

**12.** Opisati detaljno primenu okna StackPane?

🡪 **StackPane** predstavlja okno koje postavlja čvorove u centar, a njih stavlja jedan na drugi. Promenom dimenzija čvorova, čvorovi ostaju u centru jer **StackPane** automatski postavlja čvorove u centar.

**13.** Opisati detaljno primenu okna GridPane?

🡪 **GridPane** predstavlja okno koje uređuje čvorove u **matričnom** obliku. Čvorovi se postavljaju u ćelije matrice (kolone i redovi).

**14.** Opisati detaljno primenu okana HBox i VBox?

🡪 **HBox** predstavlja okno koje sve čvorove stavlja u jedan **horizontalni red**.

🡪 **VBox** predstavlja okno koje sve čvorove stavlja u jednu **vertikalnu kolonu**.

**15.** Klasa Shapes i njene podklase? Čemu služe i kako se koriste?

🡪 Klasa **Shape** je apstraktna klasa koja definiše zajednička svojstva za sve **oblike**. Podklase klase Shape su: **Text**, **Line**, **Rectangle**, **Circle**, **Ellipse**, **Arc**, **Polygon** i **Polyline**.

**16.** Objasnite tipičan scenario rada sa događajima?

🡪 Tipičan scenario rada sa događajima:

* Korisnik klikne određeno dugme na **grafičkom interfejsu**
* Kao rezultat, program generiše odgovarajući signal - tj. određeni **tip događaja** (**event**)
* Softverski objekat - **obrađivač događaja** (**event handler**) - obrađuje događaj time što inicira odgovarajuću akciju programa

**17.** Šta je izvor događaja? Šta je događaj? Šta je obrađivač događaja?

🡪 **Izvor događaja** je **objekat** koji kreira i “ispaljuje” neki događaj. Taj objekat se drugačije naziva **izvorni objekat** ili **izvorna komponenta**.

🡪 **Događaj** (**event**) predstavlja signal programu da se nešto desilo. To je **objekat** koji je kreirao izvorni objekat. Primer događaja jeste rad sa mišem ili kucanje po tastaturi.

🡪 **Obrađivač događaja** (**osluškivač događaja**) jeste **objekat** koji obrađuje događaj. Java koristi **model delegiranja** za obrađivanje događaja.

**18.** Šta je EventHandler? Šta je klasa obrađivač događaja?

🡪 **EventHandler** je interfejs za obradu događaja **EventHandler<T extends Event>** za neki događaj T. Ovaj interfejs sadrži metod **handle (T event)** za njihovu obradu.

🡪 **Obrađivač događaja** je objekat koji mora da se registruje kod izvornog objekta događaja i mora da bude primerak odgovarajućeg interfejsa za obrađivanje događaja. Za ActionEvent tip događaja metod je **setOnAction()**. Za događaj koji nastaje pritiskom dugmeta miša, koristi se metod **setOnMousePressed()**. Za događaj koji nastaje pritiskom dugmeta na tastaturi, metod za obradu je **setOnKeyPressed()**.

**19.** Šta je unutrašnja klasa? Šta je unutrašnja anonimna klasa, a šta je lambda izraz?

🡪 **Unutrašnja klasa**, ili povezana klasa, je klasa koja je definisana unutar neke druge klase. **Unutrašnje klase** su korisne za definisanje klase obrađivača događaja.

🡪 **Anonimna unutrašnja klasa** je unutrašnja klasa bez naziva. Kombinuje definisanje unutrašnje klase i kreiranje primerka (objekta) te klase, u jednom koraku.

🡪 **Lambda izraz** je anonimna klasa sa pojednostavljenom sintaksom. Primenom **lambda izraza** može se znatno uprostiti programiranje postupka rada sa događajima.

**20.** Objasnite registraciju obrađivača događaja i proces obrade događaja.

🡪 Korisnik izvrši akciju i ispaljuje događaj - **Event**. Kreira se osluškivač **setOnEventType** koji registruje događaj, a **EventHandler** obrađuje događaj.

**21.** Da bi neki objekat bio obrađivač događaja, mora da zadovolji dva preduslova, koja?

🡪 **EventHandler** je interfejs za obradu događaja **EventHandler<T extends Event>** za neki događaj T. Ovaj interfejs sadrži metod **handle (T event)** za njihovu obradu.

🡪 **Obrađivač događaja** je objekat koji mora da se registruje kod izvornog objekta događaja i mora da bude primerak odgovarajućeg interfejsa za obrađivanje događaja.

**22.** Koje različite tipove događaja poznajete?

🡪 Različiti **tipovi događaja**:

* ActionEvent događaj 🡪 metod: **setOnAction()**
* događaj koji nastaje pritiskom dugmeta miša 🡪 metod: **setOnMousePressed()**
* događaj koji nastaje pritiskom dugmeta na tastaturi 🡪 metod: **setOnKeyPressed()**

**23.** Kako se osluškuju i obrađuju događaji GUI kontrola?

🡪 **GUI kontrole** ispaljuju događaj tipa **ActionEvent** koji osluškujemo sa **setOnAction()**, a obrađujemo sa **EventHandler**.

**24.** Kako se osluškuju i obrađuju događaji miša?

🡪 Objekat **MouseEvent** sadrži informacije o događaju miša, kao što su broj klikova, lokaciju miša (x i y koordinate) kao i koje je dugme pritisnuto. **MouseEvent** se javlja uvek kada se pritisne, otpusti i klikne dugme miša, kao i kada se miš kreće ili kada se scena pomera mišem. Četiri konstante: **PRIMARY**, **SECONDARY**, **MIDDLE** i **NONE** koje su definisane u klasi **MouseButton**, označavaju levo, desno, srednje dugme miša, kao i odsustvo miša. Korišćenjem metoda **getButton()**, može se utvrditi koje je dugme pritisnuto.

**25.** Kako se osluškuju i obrađuju događaji tastature?

🡪 **KeyEvent** se javlja uvek kada dođe do pritiska, opuštanja i kucanja tastera nekog čvora ili scene. Objekat **KeyEvent** opisuje prirodu događaja (označava koji je taster pritisnut, opušten ili otkucan) i vrednost koda tastature. Svaki događaj tastera vraća metod **getCode()**. Kodovi tastera su konstante u klasi **KeyCode**.

**26.** Šta je animacija? Koje vrste animacija u JavaFX platformi poznajete?

🡪 U JavaFX-u, **animacija** predstavlja promenu svojstva čvorova tokom vremena. **Animacija** uvodi novu dimenziju - **vreme**. Scena sa grafičkim objektima se menja sa vremenom. Neke od najpoznatijih animacija su: **PathTransition**, **FadeTransition**, **RotateTransition**, **ScaleTransition** i **TranslateTransition**.

**27.** Koje osobine poseduje svaka animacija? Kako se pokreće, pauzira i zaustavlja animacija?

* Svojstvo **autoReverse** je Boolean promenljiva koja označava da li se animacija odvija u suprotnom smeru u sledećem ciklusu.
* Svojstvo **cycleCount** označava broj ciklusa animacije.
* Svojstvo **rate** definiše brzinu animacije.
* **Negativna vrednost** označava suprotni smer animacije.
* Metodi **play()**, **pause()** i **stop()** čine da se animacija pokrene, privremeno ili stalno zaustavi.

**28.** Objasnite animaciju po kojoj se objekat kreće po obliku određenom drugim objektom.

🡪 Reč je o **PathTransition** animaciji. **PathTransition** animira kretanje čvora duž nekog puta od jednog do drugog kraja u određenom vremenskom periodu. Klasa **Duration** definiše vremensko trajanje animacije. Metod **compareTo()** služi za upoređivanje vremenskih trajanja animacija. Konstante **NONE** i **ORTHOGONAL\_TO\_TANGENT** određuju da li je položaj čvora normalan u odnosu na tangentu geometrijskog puta.

**29.** Objasnite animaciju po kojoj se objekat pojavljuje i nestaje sa scene.

🡪 Reč je o **FadeTransition** animaciji. **FadeTransition** vrši animaciju promene neprozirnosti čvora u nekom vremenskom periodu. Početna i krajnja vrednost prozirnosti objekta se poševama metodama **setFromValue(0.0)** i **setToValue(1.0)**. Može se i podesiti da se animacija odigrava beskonačno: **setCycleCount(Timeline.INDEFINITE)**.

**30.** U čemu je razlika između Labeled i Label?

🡪 Klasa **Label** kreira natpis u kome se prikazuje tekst, neki čvor, ili oba. Klasa **Labeled** sadrži zajednička svojstva koja dele klase Label, Button, CheckBox i RadioButton. **Label** se obično koristi za označavanje drugih UI kontrola (najčešće **TextField**).

**31.** Kako se kreira dugme? Šta se postiže primenom ove kontrole?

🡪 **Dugme** (klasa **Button**) je kontrola koja pokreće jedan događaj akcije kada se klikne. Zajednička svojstva dugmadi su definisana u klasama **ButtonBase** i **Labeled**. Dugme ima svojstvo **onAction** koje je definisano u klasi **ButtonBase**. Ovaj metod postavlja obrađivača za obradu akcije dugmeta.

**32.** Kojim kontrolama je omogućen unos teksta? Kako se kreiraju objekti navedenih kontrola?

🡪 Unos teksta je omogućen klasama **TextField** i **TextArea**. Za unos teksta u jednom redu koristi se klasa **TextField**. Za unos teksta u više redova, koristi se klasa **TextArea**. Klasa **TextArea** obezbeđuje korišćenje klizača (za pomeranje teksta), ali je korisno da se kreira objekat **ScrollPane** u kom se smeštaju primerci klasa **TextArea** i **ScrollPane**.

**33.** Objasnite čemu služe kontrole ListView i ComboBox? U čemu se ove kontrole suštinski razlikuju?

🡪 Klasa **ComboBox** obezbeđuje padajuću listu sa opcijama koje korisnik može da izabere.

🡪 Klasa **ListView** daje prikaz liste, kao što daje i **ComboBox** klasa, s tom razlikom što korisnik, pored jedne ponuđene stavke, može da izabere i više ponuđenih stavki.

**34.** Šta je ScrollBar, a šta Slider?

🡪 Klasa **ScrollBar** predstavlja kontrolu koja omogućava korisniku da izabere neku vrednost u okviru navedenog opsega mogućih vrednosti.

🡪 Klasa **Slider** omogućava korisniku unos vrednosti primenom klizača, tj. trake sa klizačem. Klasa **Slider** je slična klasi **ScrollBar**, samo pruža više svojstava i omogućava više oblika prikazivanja.

**35.** Objasnite primenu klasa Media, MediaPlayer i MediaView?

🡪 Klasa **Media** određuje izvor medije. Media predstavlja izvor medija sa svojstvima kao što su trajanje, širina i visina. **Media** objekat se može kreirati iz URL stringa.

🡪 Klasa **MediaPlayer** emituje i kontroliše mediju (video ili audio prikaz). Ima značajna svojstva poput: autoPlay, currentCount, cycleCount, mute, volume i totalDuration.

🡪 Klasa **MediaView** obezbeđuje svojstva koja omogućavaju prikaz (gledanje i slušanje) medija, tj. video i audio prikaza. **MediaView** prikazuje objekat **Media** koji je u izvršenju od strane objekta **MediaPlayer**.

**GRUPA PITANJA 2**

**36.** Šta je rekurzija?

🡪 **Rekurzija** je metoda koja poziva samu sebe. **Rekurzija** često pojednostavljuje rešavanje određenih problema. Svaki problem koji se može rešiti **rekurzijom**, može se rešiti i **petljom**. Ipak, **rekurzija** je tehnika koja na elegantan način rešava probleme koji se teško rešavaju upotrebom **petlji**.

**37.** Šta podrazumeva rekurzivni način pristupa rešavanju problema?

🡪 **Rekurzivni način** pristupa rešavanju problema podrazumeva da imamo metodu sa **rekurzivnim pozivom**, tj. metod koji poziva samog sebe. **Rekurzivne** metode imaju sledeće karakteristike:

* Koristi se if-else ili switch iskaz koji dovodi do različitih slučajeva.
* Postoji jedan ili više osnovnih slučajeva za zaustavljanje rekurzije.
* Svaki rekurzivni poziv smanjuje početni problem, dovodeći ga bliže osnovnom slučaju dok ne postane takav slučaj.

Da bi se neki problem rešio **rekurzijom**, on se mora razdvojiti na potprobleme dok se ne dođe do osnovnog slučaja.

**38.** Navedite prednosti i nedostatke primene rekurzije?

🡪 **Prednost** rekurzije je to što u nekim slučajevima dobijamo znatno kraće, brže i preglednije rešenje u odnosu na iterativni način rešavanja problema.

🡪 **Nedostatak** rekurzije je to što zahteva dosta računarskih resursa, jer svaki put kada program poziva metodu, on mora obezbediti prostor za sve lokalne promenljive i parametre.

**39.** Kada koristite rekurziju, a kada petlju?

🡪 Korišćenje **rekurzije** ili **petlje** zavisi od prirode problema, kao i od razumevanja problema i ideje kako ga treba rešiti. Po pravilu, bira se pristup koji nam se intuitivno nameće, a koji odražava problem. Po pravilu, **iterativne** metode su efikasnije, i ako je njihova primena jasna u određenom problemu, treba ih koristiti. Svakako, ukoliko se neki problem elegantno i brzo može rešiti **rekurzijom**, onda je treba iskoristiti.

**40.** Šta je repna rekurzija?

🡪 **Repna rekurzija** (**tail recursion**) predstavlja rekurziju koja nema zaostale operacije na čekanju, a koje bi trebalo izvršiti po završetku jednog rekurzivnog poziva. **Repna rekurzija** je poželjna, jer se metod završava odmah kada se završi i poslednji rekurzivni poziv.

**41.** Šta je i kako se koristi pomoćna rekurzivna metoda?

🡪 **Rekurzivni pomoćni metodi** su vrlo korisni za nalaženje rekurzivnih rešenja problema rada sa tekstovima i nizovima. Može se definisati novi metod **pomoćne rekurzije** sa pomoćnim parametrima. Na ovaj način se redefiniše originalni metod sa istim imenom ali sa različitim potpisom.

**42.** Šta je uslov za zaustavljanje rekurzije? Da li je obavezno njegovo korišćenje?

🡪 **Uslov za zaustavljanje rekurzije** jeste osnovni slučaj do kojeg se dolazi rekurzijom. On je **obavezan** u svakoj rekurzivnoj metodi kako bi mogla da dođe do kraja izvršavanja, inače će metoda pozivati samu sebe u nedogled (**infinite loop**).

**43.** Šta su generičke klase?

🡪 **Generičke klase** omogućavaju parametrizovanje tipova. To znači da se može definisati klasa ili metod sa **generičkim** (opštim) tipom koje kompajler može da zameni sa **konkretnim** tipovima. Primera radi, definisanjem **generičke** klase ArrayList, može se kreirati objekat ArrayList koji sadrži stringove, ali i objekat ArrayList koji sadrži brojeve. Stringovi i brojevi su **konkretni** tipovi koji zamenjuju **generičke** (opšte) tipove.

**44.** Navedite prednosti primene generičkih klasa i metoda?

🡪 Najveća **prednost** primene **generičkih** klasa i metoda je u pronalaženju grešaka u vreme kompilacije umesto u fazi izvršenja programa. **Generička** klasa ili metod omogućava parametrizovanje tipova i dozvoljava specificiranje dozvoljenih tipova objekata sa kojima klasa ili metod može da radi. Takođe, **generici** eliminišu konverziju tipa (casting).

**45.** Kako se definišu generičke klase i interfejsi?

🡪 **Generički tip** se definiše za klasu ili interfejs. Kada se klasa koristi pri kreiranju objekta, mora da se koristi i **konkretan** tip.

*public class Generic<E> {*

*private E object;*

*public void set(E o) {*

*this.object = o;*

*}*

*public E get() {*

*return this.object;*

*}*

*}*

**E** 🡪 bilo koji objekat

**46.** Šta je generička metoda i kako se definiše?

🡪 **Generička metoda** je metoda koja sadrži generički tip <**E**> ili <**T**> nakon ključnih reči u zaglavlju metode.

*public static <E> void showElements(E… elements)*

**47.** Šta je sirovi tip? Šta podrazumeva pojam kompatibilnost unazad?

🡪 **Sirovi tip** je generička klasa koja se koristi bez konkretnog tipa parametra. Upotreba **sirovih tipova** omogućava **kompatibilnost** unazad, tj. **kompatibilnost** sa ranijim verzijama Jave.

**48.** Objasnite primenu generičkih džoker tipova. Koje generičke džoker tipove poznajete?

🡪 **Generički džoker tipovi** se koriste da bi se specificirao opseg nekog generičkog tipa. Postoje tri vrste džokera:

* ? ili ? extends Object 🡪 **neograničeni džoker**
* ? extends T 🡪 **ograničeni džoker**
* ? super T 🡪 **džoker sa donjom granicom**

**49.** Šta znači brisanje generičkih tipova?

🡪 Informacija o genericima koji se upotrebljavaju za vreme kompilacije, a koji nisu raspoloživi u vreme izvršenja jeste **brisanje tipa**. Pristup **brisanja tipa** omogućava generičkom kodu da ostvari kompatibilnost unazad sa starim kodom koji upotrebljava sirove tipove.

**50.** Navedite ograničenja primene generičkih tipova podataka.

🡪 Upotrebu **generičkih tipova** prate i određena **ograničenja** o kojima se treba voditi računa:

* Ne može se kreirati instanca (objekat) upotrebom parametra generičkog tipa.
* Ne može se kreirati niz upotrebom parametra generičkog tipa.
* Parametar generičkog tipa neke klase nije dozvoljen u statičkom kontekstu.
* Klase izuzetka ne mogu biti generičke.
* Ne mogu se primeniti generički tipovi sa primitivnim tipovima.
* Ne može se upotrebiti konverzija tipova.
* Ne može se preopteretiti (overload) metod sa formalnim tipovima parametara.

**51.** Zašto koristimo generičke tipove podataka kod kolekcija: lista, redova, setova...?

🡪 Kod **kolekcija**, **generički tipovi** podataka se mogu koristiti kako bi mogli u **kolekciju** da stavimo bilo koji objekat koji nam je potreban u programu.

**52.** Šta predstavlja Java Collection Framework?

🡪 **Java Collection Framework** (**JCF**) je **kolekcija struktura podataka** u Javi. Ovu **kolekciju** čine: **liste**, **vektori**, **stekovi**, **redovi** i **prioritetni redovi**. **Struktura podataka** je kolekcija (zbirka) podataka organizovanih na neki način a podržava i operacije neophodne za pristup podacima i za rad sa njima. **JCF** podržava dva tipa kontejnera: **kolekcija** i **mapa**. Postoji više vrsta kolekcija: **skupovi**, **liste** i **redovi**.

**53.** Šta je iterator i zašto ga koristimo?

🡪 **Iterator** je objekat koji se koristi da bi se pristupilo svakom od elemenata kolekcije. Uz pomoć **iteratora**, pretražuje se neka struktura podataka bez potrebe da se ulazi u detalje kako su podaci uskladišteni u strukturi podataka.

**54.** Šta je List? Koje liste poznajete i po čemu se razlikuju?

🡪 **Lista** predstavlja **kolekciju** za smeštaj elemenata u sekvencijalnom rasporedu. Interfejs **List** nasleđuje interfejs **Collection**. **Liste** se kreiraju korišćenjem klasa **ArrayList** ili **LinkedList**. **ArrayList** smešta elemente u niz koji se dinamički kreira. **LinkedList** smešta elemente u vidu povezane liste.

**55.** Šta je omogućeno implementacijom interfejsa Comparator i Comparable? Koje tipove podataka implementiraju metode ovih interfejsa?

🡪 **Comparator** se koristi za upoređivanje objekata različitih klasa. Koristimo ga u klasama koje ne implementiraju interfejs **Comparable**. Klasa mora da implementira interfejs **Comparator <T>**, koji ima dve metode, **compare()** i **equals()**. Metod **compare()** je **int** tipa, vraća negativnu vrednost ako je prvi element manji od drugog, pozitivnu vrednost ako je prvi element veći od drugog i nulu ako su oba elementa jednaka. Metod **equals()** je **boolean** tipa i vraća true ako su oba elementa jednaka.

🡪 **Comparable** se koristi za upoređivanje objekata iste klase. Comparable definiše metod **compareTo()**, koji služi za upoređivanje dva elementa klasa koje implementiraju interfejs **Comparable**.

**56.** Da li je moguće koristiti korisne statičke metode za liste i kolekcije? Objasnite, ukoliko je moguće, koja klasa je zadužena za poziv navedenih metoda?

🡪 Klasa **Collections** sadrži statičke metode radi izvršenja zajedničkih operacija i u kolekciji i u listi. Neke od ovih metoda su: **sort**, **binarySearch**, **reverse**, **shuffle**, **copy**, **fill**, **max**, **min**, **disjoint**, **frequency**.

**57.** Šta je Vector, a šta Stack?

🡪 Klasa **Vector** je ista kao i klasa **ArrayList**, sem što poseduje i metode za sinhronizaciju pristupa i promenu objekata klase **Vector**. Ovi metodi sprečavaju oštećenje podataka kada se pristupa **vektoru** i kada se menja dejstvo dva ili više simultana toka.

🡪 Klasa **Stack** predstavlja strukturu podataka koja funkcioniše po principu **last in - first out**. To znači da prvi iz steka izlazi onaj koji je poslednji ubačen. Klasa **Stack** je podklasa klase **Vector**. Metod **peak()** pristupa elementu na vrhu steka, bez njegovog uklanjanja. Metod **pop()** uklanja element na vrhu steka i vraća taj element. Metod **push(Object element)** dodaje element na stek.

**58.** Šta je red? Koje vrste redova poznajete? Kako se kreiraju i šta im je namena?

🡪 **Red** (**queue**) je struktura podataka koja primenjuje princip **FIFO** (**first in - first out**). Elementi se dodaju na kraj reda, a uklanjaju sa početka reda. Pored **reda**, postoje i **prioritetni redovi** (**priority queue**). Kod njih se elementima dodeljuje **prioritet**. Kada se pristupa elementima, prvo se uklanja iz reda element koji ima najveći **prioritet**.

**59.** Napravite razliku između struktura red i stek.

🡪 **Red** funkcioniše po principu **FIFO** (**first in - first out**), tj. onaj koji je prvi stavljen u **red**, prvi izlazi iz njega.

🡪 **Stek** funkcioniše po principu **LIFO** (**last in - first out**), tj. onaj koji je poslednji ubačen u **stek**, prvi izlazi iz njega.

**60.** Po čemu se razlikuje prioritetni red od “običnog’’?

🡪 **Prioritetni red** za razliku od običnog **reda** ređa svoje elemente po **prioritetu**, tj. njihovoj vrednosti. Tako najveći **prioritet** imaju oni elementi sa najmanjom vrednošću. Ukoliko više elemenata imaju isti **prioritet**, onda izlaze po slučajnom rasporedu.

**GRUPA PITANJA 3**

**61.** Šta je set? Koje vrste setova poznajete?

🡪 **Set** (**skup**) predstavlja **strukturu podataka** za skladištenje elemenata bez njihovog **dupliranja** (**ponavljanja**). Interfejs **Set** proširuje interfejs **Collection**. Postoje tri vrste seta: **HashSet**, **LinkedHashSet** i **TreeSet**.

**62.** U čemu je suštinska razlika seta i liste?

🡪 **Setovi** su efikasniji od **listi** za skladištenje elemenata bez njihovog dupliranja. **Liste** su korisne za pristup elementima preko indeksa. **Setovi** ne podržavaju indekse, jer elementi nisu smešteni po nekom redosledu.

**63.** Šta je heširanje?

🡪 **Heširanje** (**hashing**) je tehnika koja nalazi vrednost u nekom nizu upotrebom **indeksa**, tj. **heš koda**, koji se dobija iz ključa, bez sprovođenja istraživanja.

**64.** Diskutujte performanse setova i listi?

🡪 **Setovi** su mnogo efikasniji od **listi** kod testa da li je neki broj u setu ili listi. Razlog je u **implementaciji** listi i setova. **Setovi** su efikasniji od **listi** za skladištenje elemenata bez njihovog ponavljanja. **Liste** su korisne za pristup elementima preko indeksa.

**65.** Šta je mapa? Šta omogućava interfejs Map?

🡪 **Mapa** je kontejner objekat koji skladišti kolekciju parova **ključ / vrednost**. **Mapa** omogućava brzu pretragu, brisanje i menjanje parova primenom ključa pretrage. **Mapa** skladišti vrednosti zajedno sa ključevima. **Ključevi** su kao indeksi, i to mogu biti bilo kakvi objekti. **Interfejs Map** obezbeđuje metode za pretraživanje, menjanje i dobijanje kolekcija vrednosti i seta ključeva.

**66.** Koje vrste mapa poznajete? Kako se kreiraju objekti mapa?

🡪 Postoje tri vrste mapa: **HashMap** (**heš mapa**), **LinkedHashMap** (**povezana heš mapa**) i **TreeMap** (**mapa stabla**). Zajednička svojstva ovih mapa su definisana u interfejsu **Map**. Kreiranje objekata mapa:

* *Map<String, Integer> hashMap = new HashMap<>();*
* *Map<String, Integer> linkedHashMap = new LinkedHashMap<>();*
* *Map<String, Integer> treeMap = new TreeMap<>();*

**67.** Šta je nit, a šta zadatak?

🡪 **Nit** je tok izvršenja jednog zadatka, od početka do kraja. **Nit** predstavlja objekat interfejsa **Runnable** koji omogućava izvršenje jednog zadatka.

🡪 **Zadatak** predstavlja objekat. Klasa zadatka mora da primeni interfejs **Runnable**. Svaki zadatak mora da se izvršava pod kontrolom jedne niti.

**68.** Kako se kreira nit, a kako zadatak?

🡪 Kreiranje **zadatka**: *Task task = new Task();*

🡪 Kreiranje **niti**: *Thread thread = new Thread(task); thread.start();*

**69.** U čemu se razlikuju pozivi metoda run() i start()?

🡪 **Metoda run()** se implementira iz interfejsa **Runnable** i u njoj definišemo kako želimo da se naš zadatak ponaša. Kada kreiramo nit sa zadatkom, nit se startuje **metodom start()**.

**70.** Objasnite primenu metoda klase Thread za upravljanje izvršavanjem niti?

* **run()** 🡪 definiše se šta zadatak treba da uradi (ponašanje zadatka)
* **start()** 🡪 startovanje niti
* **yield()** 🡪 vreme procesora se privremeno oslobađa za druge niti
* **sleep()** 🡪 stavljanje niti u stanje spavanja određeni broj milisekundi da bi se dozvolilo drugim nitima da se izvršavaju
* **join()** 🡪 obezbeđuje da jedna nit čeka da se druga nit izvrši
* **setPriority()** 🡪 određivanje prioriteta niti
* **getPriority()** 🡪 dobijanje prioriteta niti

**71.** Šta je pul niti?

🡪 Upotreba **pula niti** je idealan način za istovremeni rad sa puno zadataka. Java obezbeđuje interfejs **Executor** za izvršenje zadataka u jednom **pulu niti** i interfejs **ExecutorService** za upravljanje i kontrolu zadataka.

**72.** Po čemu se razlikuju interfejsi Executor i ExecutorService?

🡪 Java obezbeđuje interfejs **Executor** za izvršenje zadataka u jednom **pulu niti** i interfejs **ExecutorService** za upravljanje i kontrolu zadataka. **ExecutorService** je podinterfejs interfejsa **Executor**.

**73.** Šta je sinhronizacija niti i zašto je potrebno obaviti sinhronizaciju?

🡪 **Sinhronizacija niti** je koordinacija izvršenja zavisnih niti. **Sinhronizacija niti** se obavlja da ne bi došlo do **race condition-a** (konflikt između dve niti koje istovremeno pristupaju zajedničkom resursu). Klasa je bezbedna za nit (**thread-safe**) ako ne dovodi do stvaranja race condition-a.

**74.** Kada koristimo rezervisanu reč synchronized?

🡪 Rezervisanu reč **synchronized** možemo iskoristiti da bi sinhronizovali metod tako da samo jedna nit može da pristupi resursu vezanom za tu ključnu reč u nekom vremenskom trenutku. Sinhronizovani metod zahteva **zaključavanje** (**lock**) pre izvršenja. **Zaključavanje** je mehanizam za ekskluzivno korišćenje resursa.

**75.** Na koje načine je moguće obaviti sinhronizaciju niti?

🡪 **Sinhronizacija niti** se može obaviti pomoću ključne reči **synchronized** i **upotrebom ključeva**. **Ključevi** i **uslovi** se mogu eksplicitno koristiti za **sinhronizaciju niti**. **Ključnu reč** koristimo tako da samo jedna nit može da pristupi resursu vezanom za tu ključnu reč u nekom vremenskom trenutku. Java omogućava da se eksplicitno zahteva ključ, što daje više kontrole traženja i oslobađanja ključeva. Interfejs **Lock** koristi metod **newCondition()** za kreiranje bilo kog broja **Condition** objekata, koji mogu da budu upotrebljeni za komunikaciju sa nitima.

**76.** U čemu se razlikuju pozivi metoda newFixedThreadPool(int) i newCachedThreadPool()?

🡪 **newFixedThreadPool(int)** kreira fiksni broj niti u pulu. Ako nit završi izvršenje zadatka, može se ponovo koristiti za neki drugi zadatak.

🡪 **newCachedThreadPool** kreira novu nit ako nisu sve niti u pulu van rada i ako ima zadataka koji čekaju na izvršenje.

**77.** Objasnite sinhronizaciju upotrebom ključeva?

🡪 **Sinhronizacija upotrebom ključeva**: Kreira se instanca interfejsa **Lock**, tj. njegova konkretna primena **ReentrantLock**. U metodi koju želimo da sinhronizujemo, na samom početku se zaključava upotrebom metode **lock()**, a nakon što se dođe do kraja metode, otključava se pozivom metode **unlock()**.

**78.** Šta predstavlja i kako se realizuje saradnja (kooperacija) između niti?

🡪 **Kooperacija** između niti se može ostvariti koordinacijom rada sa ključevima. Koordinacija se realizuje poštovanjem definisanih uslova za rad niti. Saradnja se realizuje korišćenjem klase *Condition uslov = lock.newCondition();* .

**79.** Šta je blokada ključeva i kako se ona sprečava?

🡪 **Blokada ključeva** je pojava kada dve ili više niti zahtevaju ključeve za pristup objektima koje zajednički koriste. **Blokada ključeva** se sprečava upotrebom jednostavne tehnike poznate kao **redosled resursa**. Ovom tehnikom, određuje se redosled svih objekata čiji se ključevi traže da bi obezbedili da svaka nit dobija ključ po tom rasporedu.

**80.** Šta je blokirajući red i šta je njegova namena?

🡪 **Blokirajući red** je red koji čini da nit blokira unos elementa u puni red ili da blokira uklanjanje elementa iz praznog reda. Interfejs **BlockingQueue** obezbeđuje sinhronizovane metode **put()** i **take()** za dodavanje i uklanjanje elemenata.

**81.** Koje tipove blokirajućih redova poznajete?

🡪 Za podršku blokirajućih redova, Java obezbeđuje tri klase za tri tipa blokirajućih redova:

* **ArrayBlockingQueue** 🡪 tip blokirajućeg reda koji upotrebljava **niz** (**array**)
* **LinkedBlockingQueue** 🡪 tip blokirajućeg reda koji upotrebljava **povezanu listu** (**linked list**)
* **PriorityBlockingQueue** 🡪 tip blokirajućeg reda koji upotrebljava **prioritetni red** (**priority queue**)

**82.** Šta je semafor i šta je njegova namena?

🡪 **Semafor** je objekat koji kontroliše pristup zajedničkim resursima. **Semafori** se koriste za ograničenje broja niti koje mogu da pristupe deljivom resursu.

**83.** Koja stanja nit može da ima?

🡪 **Stanje niti** ukazuje na **status niti**. Niti mogu biti u jednom od pet stanja:

* **New** 🡪 početno stanje kod novokreiranih niti
* **Ready** 🡪 dejstvom metode start(), nit počinje da radi
* **Running** 🡪 spremna nit počne izvršenje
* **Blocked** 🡪 nit postaje neaktivna (poziv metoda join(), sleep(), wait() ili čeka završetak I/O operacije)
* **Finished** 🡪 nit je u celosti izvršila svoj zadatak

**84.** Kako se i iz kojeg razloga sinhronizuju kolekcije?

🡪 **Java Collections Framework** obezbeđuje **sinhronizovane kolekcije** za **liste**, **setove** i **mape**. Sadržaj ovih klasa može da bude oštećen ako im se pristupa istovremeno od strane više niti. Podaci u kolekciji se mogu zaštititi **zaključavanjem kolekcije** ili upotrebom **sinhronizovanih kolekcija**. Klasa Collections obezbeđuje šest statičkih metoda za umotovanje kolekcije u sinhronizovanu verziju (kolekcije 🡪 **sinhronizovani omotači**).

**85.** Šta je paralelno programiranje?

🡪 **Paralelno programiranje** je tehnika programiranja koja omogućava korišćenje više niti koje se paralelno izvršavaju rešavajući nezavisne potprobleme problema za koji se program pravi.

**86.** Šta je ForkJoinTask, a šta ForkJoinPool?

🡪 **ForkJoinTask** definiše jedan zadatak. Predstavlja apstraktnu klasu za sve zadatke.

🡪 **ForkJoinPool** upravlja radom nitima i pruža alate za dobijanje informacija o stanju i performansama rada niti.

**GRUPA PITANJA 4**

**87.** Šta čini sistem relacione baze podataka?

🡪 **Sistem relacione baze podataka** čini baza podataka, softver za skladištenje i upravljanje podacima u bazi podataka, aplikacioni programi koji koriste podatke i njihovi korisnici.

**88.** Koji je standardan scenario obavljanja operacije nad bazom podataka iz Java programa?

🡪 Standardan scenario obavljanja operacije nad bazom podataka se sastoji od nekoliko sledećih koraka:

* Instaliranje **JDBC** uslužnih programa (**drajvera**)
* Uspostavljanje **veze** (**konekcije**)
* Kreiranje **SQL** iskaza
* Izvršavanje **SQL** iskaza
* Obrada **ResultSet-a**

**89.** Šta je JDBC?

🡪 **JDBC** je **Java API** koji se koristi za razvoj Java aplikacija koje koriste relacione baze podataka. **JDBC** obezbeđuje programerima u Javi jedan uniformni interfejs za pristup i za rad sa relacionim bazama podataka. Upotrebom **JDBC ApI**, aplikacije pisane u Javi mogu da izvrše SQL iskaze, da dobijaju rezultate, da predstave podatak na način koji je ugodan korisniku i da vrate promenjene podatke u bazu podataka.

**90.** Šta predstavlja drajver baze podataka?

🡪 **JDBC uslužni program** (**driver**) služi kao interfejs koji olakšava komunikaciju između JDBC i neke posebne baze podataka. JDBC uslužni programi se razvijaju za specifične baze podataka, tj. **DBMS**.

**91.** Šta je RDBMS?

🡪 **RDBMS** (**Relational Database Management System**) predstavlja sistem za upravljanje relacionim bazama podataka. Ovaj sistem omogućava kreiranje, održavanje i kontrolu baze podataka.

**92.** Šta čini relacioni model baze podataka?

🡪 **Relacioni model baze podataka** čine tri komponente: struktura, integritet (celovitost) i jezik.

* **Struktura** definiše predstavljanje podataka.
* **Integritet** uvodi ograničenja podacima.
* **Jezik** obezbeđuje način pristupa i manipulacije sa podacima.

**93.** Šta je struktura baze podataka?

🡪 **Struktura** baze podataka definiše predstavljanje podataka.

**94.** Šta je integritet baze podataka? Koja ograničenja integriteta postoje?

🡪 **Integritet** baze podataka uvodi ograničenja podacima. Svako ograničenje integriteta definiše uslov koje vrednosti u nekoj tabeli moraju da zadovolje. Postoje tri tipa ograničenja:

* Ograničenja **domena**
* Ograničenja **primarnog ključa**
* Ograničenja **sekundarnog ključa**

Ograničenja domena i primarnog ključa su **intra-relaciona ograničenja** jer se odnose samo na jednu relaciju (tabelu). Ograničenje sekundarnog ključa je **inter-relaciono**, jer se odnosi na više od jedne relacije (tabele).

**95.** Šta je jezik baze podataka?

🡪 **Jezik** baze podataka obezbeđuje način pristupa i manipulacije sa podacima. **SQL** je univerzalni jezik za pristupanje relacionim sistemima baza podataka.

**96.** Koji tipovi SQL upita postoje (osnovne dve grupe)? U zavisnosti od tipa upita, koja metoda se poziva kada joj se kao String argument predaje SQL upit?

🡪 **Prva grupa SQL upita** je za kreiranje baze i njene definicije, a **druga grupa upita** je za manipulaciju podataka u bazi. SQL upit se može izvršiti upotrebom **executeQuery** (String sql).

**97.** Koje JDBC interfejse poznajete i šta je njihov zadatak?

🡪 Četiri ključna **JDBC interfejsa** neophodna za razvoj aplikacije sa bazom podataka su: **Driver**, **Connection**, **Statement** i **ResultSet**. Ovi interfejsi predstavljaju radni okvir za pristup generičkim SQL bazama podataka. JDBC aplikacija koristi odgovarajući JDBC uslužni program (drajver) upotrebom interfejsa **Driver**, povezuje se sa bazom podataka upotrebom interfejsa **Connection**, kreira SQL iskaz upotrebom interfejsa **Statement**, i obrađuje rezultate upotrebom interfejsa **ResultSet**, ako ih iskazi obezbeđuju.

**98.** Kada i kako se dobija ResultSet objekat? Kako se obavlja izvlačenje rezultata iz ovog objekta?

🡪 Klasa **ResultSet** sa metodom **next()** čita red po red kolona koje su dobijene kao rezultat izvršenja **SQL upita**. Klasa **ResultSet** održava tabelu čiji se red čita. Početna pozicija reda je **null**. Pored metoda next(), koriste se i različiti **getter** metodi za dobijanje vrednosti iz tekućeg reda.

*Statement statement = connection.createStatement();*

*ResultSet resultSet = statement.executeQuery(String query);*

**99.** Šta je PreparedStatement? Po čemu ovaj tip Statement objekta ima najčešću primenu u programima?

🡪 **PreparedStatement** proširuje interfejs Statement i omogućava kreiranje parametrizovanih SQL upita. Omogućava dinamičko kreiranje SQL upita i sprečava SQL injection napade.

**100.** Šta je CallableStatement i kako se koristi?

🡪 **CallableStatement** je interfejs koji je projektovan za izvršenje memorisanih SQL procedura (SQL - stored procedures). Procedure mogu da imaju parametre: **IN**, **OUT** ili **INOUT**. CallableStatement nasleđuje PreparedStatement.

**101.** Šta su meta podaci baze podataka? Kako možemo da dobijemo ove podatke?

🡪 **Meta podaci** baze podataka, kao što su URL, ime korisnika (username), ime JDBC drajvera, dobijaju se upotrebom **DatabaseMetaData** interfejsa. Skup dobijenih meta podataka, kao što su broj kolona tabele, nazivi kolona, mogu se dobiti upotrebom **ResultSetMetaData** interfejsa. JDBC obezbeđuje interfejs **DatabaseMetaData** za dobijanje informacija o bazi podataka, a interfejs **ResultSetMetaData** za dobijanje informacija u posebnom ResultSet-u.

**102.** Šta je ORM?

🡪 **ORM** (**Object-Relational Mapping**) je tehnika programiranja koja vrši konverziju podataka između relacionih baza podataka i objektno-orijentisanih jezika, kao što su Java, C# i dr. **ORM** sistem ima veliki broj prednosti u odnosu na JDBC. Jedan od najpoznatijih Java ORM radnih okvira (framework-a) jeste **Hibernate**.

**103.** Šta je Hibernate?

🡪 **Hibernate** je softverska komponenta (**ORM framework**) koja se nalazi između Java objekata (u aplikaciji) i servera sa bazom podataka, i koja obavlja celokupan posao vezan za mapiranje objekata u relacione tabele u bazi. **Hibernate** mapira Java klase u tabele baze podataka i tipove podataka Jave u SQL tipove podataka i oslobađa programera od potrebe da realizuje 95% uobičajenih zadataka pri radu sa smeštanjem i korišćenjem objekata u relacionoj bazi podataka.

**104.** Kako se konfiguriše (podešava) Hibernate?

🡪 Objekat **Configuration** kreira **SessionFactory** objekat koji konfiguriše Hibernate za aplikaciju pri čemu koristi **konfiguracioni fajl** i dozvoljava kreiranje **Session** objekata. Configuration objekat ima dve ključne komponente:

* **Database Connection**: Ova komponenta obezbeđuje **konfiguracione fajlove** koji podržavaju Hibernate. Ovi fajlovi su **hibernate.properties** i **hibernate.cfg.xml**.
* **Klasa Mapping Setup**: Ova komponenta kreira vezu između Java klasa i tabele baze podataka.

**Session** je objekat koji se koristi za dobijanje fizičke veze sa bazom podataka. Objekat **Transaction** predstavlja jedinicu rada sa bazom podataka i sa RDBMS i podržava transakcije. **Query** realizuje upite baze, a **Criteria** definiše kriterijume.

**105.** Šta se postiže glavnim Hibernate konfiguracionim fajlom, a šta mapirajućim fajlovima za svaku entitetsku klasu?

🡪 **Glavni konfiguracioni fajl** kreira podešavanja za bazu, link do baze, jezik baze, korisničko ime i šifru. **Mapirajući fajlovi** određuju kako će se zasebne klase čuvati u bazi podataka.

**106.** Kako je moguće eliminisati XML mapirajuće fajlove primenom Java anotacija?

🡪 **Hibernate Annotations** (korišćenje **napomena**) je način za definisanje mapiranja objekata u tabele i obrnuto, bez upotrebe XML fajla. Može se koristiti kao dopuna ili kao zamena XML mapiranja meta podataka. Svi meta podaci se stavljaju zajedno u **POJO** java fajl zajedno sa programskim kodom. Korišćenjem anotacija, dobija se veća fleksibilnost nego u korišćenju XML definisanja potrebnog mapiranja.

**107.** Šta je HQL?

🡪 **HQL** (**Hibernate Query Language**) je objektno-orijentisani jezik za rad sa bazom podataka u okviru **Hibernate ORM**. **HQL** radi sa trajnim objektima i njihovim svojstvima. **HQL** upiti se prevode u uobičajene **SQL** upite (od strane Hibernate-a) koji izvršavaju potrebne akcije na bazi podataka. Ključne reči poput **SELECT**, **FROM**, **WHERE** i slično u **HQL** ne zavise od veličine slova, ali svojstva, kao što su imena tabela i kolona zavise od veličine slova.

**108.** Šta predstavljaju skraćenice TCP i UDP. Koja je razlika između njih?

🡪 Protokoli višeg nivoa su **Transmission Control Protocol** (**TCP**) i **User Datagram Protocol** (**UDP**). **TCP** omogućava da dva servera uspostave vezu i da razmene tokove podataka. **UDP** omogućava aplikaciji da pošalje datagram iz aplikacije sa jednog računara na drugi. Komunikacije sa tokovima upotrebljavaju **TCP** za prenos podataka, dok komunikacije sa paketima upotrebljavaju **UDP**. **TCP** otkriva izgubljene podatke u transmisiji, pa ih ponovo šalje. Zato se smatra pouzdanim jer garantuje prenos svih podataka, bez gubitaka. S druge strane, **UDP** ne može da garantuje da neće doći do gubljenja podataka.

**109.** Šta je Socket, a šta ServerSocket?

🡪 **Socket** predstavlja objekat klase **Socket** i to je krajnja tačka logičke veze na računaru klijenta, koja je neophodna radi komunikacije sa serverom. **Socket** je naznačen imenom ili IP adresom, preko njegovog porta, koji je označen svojim brojem.

🡪 **ServerSocket** predstavlja objekat klase **ServerSocket** koji obezbeđuje vezu sa soketom klijenta, a pri njihovom kreiranju treba se definisati broj priključka na serveru koji će soket servera koristiti.

**110.** Šta je port i čemu služi?

🡪 **Port** (**priključak**) predstavlja komponentu računara koja služi za vezu sa drugim uređajima. **Port** je podržan odgovarajućim softverskim osluškivačima njegovih veza. **Port** utvrđuje TCP servis koji je definisan u soketu.

**111.** Objasniti primenu klase InetAddress.

🡪 Serverski program upotrebljava klasu **InetAddress** za dobijanje informacije o IP adresi i nazivu računara klijenta. Korišćenjem klase **InetAddress**, može se saznati ko je sve povezan sa serverom.

**112.** Objasnite kako server radi sa više klijenata istovremeno.

🡪 Jedan serverski program može da opslužuje **više klijenata istovremeno**. Za vezu sa svakim klijentom se koristi po jedna **posebna nit** (**MultiThreadServer**). Svaka nit kreira ulazno-izlazni tok podataka (**data streams**) da bi primila i slala podatke klijentu.

**GRUPA PITANJA 5**

**113.** Šta je jedinica, a šta jedinično testiranje softvera?

🡪 **Jedinica** je klasa ili metoda koja se testira. Klasu ili metodu uvek pišemo da bude proizvod za sebe i da bi mogli da je primenimo i implementiramo u bilo kom drugom projektu.

🡪 **Jedinično testiranje** predstavlja testiranje pojedinačnih objekata. To je **test** koji testira odgovarajuću klasu, odnosno metode koje nešto rade u okviru te klase. Najbolji alat za **testiranje** pojedinačnih klasa je **JUnit**.

**114.** Navedite prednosti i nedostatke jediničnog testiranja softvera.

🡪 **Prednosti** jediničnog testiranja:

* Veći kvalitet koda
* Lakše čitanje koda
* Smanjuje količinu grešaka u kodu
* Povećava pouzdanost celog sistema
* Lako se uklapa u procedure izrade softvera

🡪 **Nedostaci** jediničnog testiranja:

* Ne može se primeniti na GUI klase
* Ne mogu da se testiraju okruženja u kojima se radi (baze podataka, mreža)
* Kod mora biti rađen po strogim OOP principima
* Sporiji razvojni ciklus
* Nekada i do 3x više koda

**115.** Šta je moguće proveriti JUnit metodama?

🡪 **JUnit metodama** se proverava:

* Tačnost rezultata za realne vrednosti
* Potencijalno problematične vrednosti (npr. deljenje sa nulom)
* Granične vrednosti
* Opseg rezultata
* Vrednosti koje su sigurno pogrešne
* Kada fali neki podatak
* Provera formata rezultata

**116.** Navedite osnovne klase koje se koriste u JUnit testovima?

🡪 **Osnovne klase** koje se koriste u okviru JUnit testova su:

* **TestCase** 🡪 predstavlja pojedinačni slučaj testiranja
* **TestSuite** 🡪 skup klasa koje se testiraju
* **TestRunner** 🡪 klasa koja obezbeđuje funkcionalnost, odnosno pokretanje testova
* **TestResults** 🡪 klasa u koju se smeštaju rezultati testa

**117.** Koje metode se nasleđuju i implementiraju unutar TestCase klase?

🡪 Metode koje se nasleđuju i implementiraju unutar **TestCase**-a su:

* **setUp()** 🡪 metoda koja nam služi da u nju postavimo osnovne postavke
* **testXXX()** 🡪 metode koje testiraju odgovarajuće metode iz konkretne klase
* **tearDown()** 🡪 metoda koja se pokreće nakon završetka testova

**118.** Šta je TestSuite?

🡪 **TestSuite** predstavlja skup klasa koje se testiraju (imamo klasu koja nasleđuje **TestSuite**, i ona u sebi može da sadrži više **TestCase**-ova).

**119.** Čemu služe assertXXX() metode? Navedite najčešće korišćene.

🡪 **Assert metode** služe za proveru ispravnosti neke vrednosti. Najčešće korišćene assert metode su:

* **assertEquals()**
* **assertNotEquals()**
* **assertTrue()**
* **assertFalse()**
* **assertSame()**
* **assertNotSame()**
* **assertThat()**
* **assertArrayEquals()**
* **assertNull()**
* **assertNotNull()**

**120.** Koji je redosled izvršavanja JUnit testova? Da li redosled može da se podešava?

🡪 Najčešće je redosled metoda koji se testiraju u test klasi slučajan, ali se može i urediti primenom **@FixMethodOrder** i korišćenjem **MethodSorters**. Primera radi, **@FixMethodOrder(MethodSorters.NAME\_ASCENDING)** sortira test metode prema **imenu** metoda, po **alfabetskom redosledu**.

**121.** Kako se testiraju izuzeci?

🡪 **Izuzeci** se testiraju tako što se kod anotacije **@Test** doda izuzetak za koji se očekuje da ga metod izbaci *@Test(expected = NullPointerException.class)*. Test će proći ukoliko metoda izbaci **očekivani izuzetak**. Kod ovakvog testiranja, assert metode nisu potrebne.

**122.** Po čemu se razlikuje assertThat() od ostalih assertXXX() metoda?

🡪 Metod **assertThat()** pojednostavljuje sintaksu za testiranje tvrdnji. Prednost primene ovog metoda jeste **čitljivija sintaksa**, ali i **čitljivije poruke o grešci**.

**123.** Kako se koriste parametrizovani testovi?

🡪 Kada se izvršavaju **parametrizovani testovi**, kreiraju se primerci sa ukrštenim elementima test metoda i test podataka. Anotacija koja stoji iznad metode jeste *@RunWith(Parameterized.class)*. Takođe, koristi se i anotacija *@Parameters*.

**124.** Šta su i kako se koriste pravila u jediničnim testovima?

🡪 **Pravila (@Rule)** dozvoljavaju fleksibilnu promenu definicije ponašanja svakog test metoda i svake test klase. Mogu da se koriste ranije definisana pravila, mogu ta ista pravila da se proširuju, ili da se definišu sasvim nova pravila.

**125.** Šta je lokalizacija i zašto se koristi u savremenom razvoju softvera?

🡪 **Lokalizacija** se uvodi zbog različitih jezika komunikacije. Mnogi softverski proizvodi se danas koriste u više zemalja, a korisnici koriste različite jezike komunikacije. Zbog toga, potrebno je da se GUI kreira na više jezika.

**126.** Šta je zadatak klase Locale? Šta sadrži ova klasa?

🡪 **Locale** predstavlja geografski, politički ili kulturni region u kome se koristi neki specifični jezik. Zadatak klase **Locale** jeste da za određeni jezik izrazi informacije kao što su datum, vreme i broj, kao i druge informacije na **lokalizovani** način, kao što je sortiranje tekstova.

**127.** Kako se identifikuje kod jezika, a kako kod države?

🡪 **Kod jezika** je malim slovima, a **kod države** velikim slovima. Primera radi, za srpski jezik, kod je **sr**, a za Srbiju - **RS**. Uz pomoć metoda **getAvailableLocals()**, mogu se dobiti sve moguće oznake podržanih zemalja i jezika.

**128.** Objasnite formatiranje datuma i vremena primenom SimpleDateFormat.

🡪 Podklasa za formatiranje datuma i vremena **SimpleDateFormat** omogućava izbor **formata** po izboru korisnika. Kada se sa konstruktorom formira objekat klase **SimpleDateFormat**, onda on konvertuje objekat **Date** u tekst (string) sa željenim **formatom**: *public SimpleDateFormat(String pattern)*. Parametar **pattern** je tekst koji čine karakteri sa specijalnim značenjem.

**129.** Na koji način je moguće podesiti format brojeva (u zavisnosti od lokalnih podešavanja)?

🡪 **Brojevi** se različito obeležavaju u različitim zemljama. Korišćenjem klase **NumberFormat** moguće je određivati i menjati **formate brojeva** u skladu sa lokalitetom. Potreban je primerak (objekat) klase **NumberFormat** da bi metodama **getInstance()** i **getNumberInstance()** dobili lokalni način prikazivanja brojeva.

**130.** Kao najveće unapređenje jezika Java, koje funkcionalnosti uvodi Java 8?

🡪 **Java 8** predstavlja najveći napredak Jave od verzije 5. **Lambda iskaz** predstavlja jednu od najvećih jezičkih promena u celoj verziji **Jave 8**. **Lambda iskaz** omogućava da tretiramo kod kao podatke i da tretiramo funkcionalnosti kao metode. Ima dosta pojednostavljeniju sintaksu. Moguće je vratiti vrednost a da pri tom se ne koristi return.

**131.** Šta je funkcionalni interfejs?

🡪 Koncept **funkcionalnog interfejsa** čini lambda iskaze **kompatibilnim**. **Funkcionalni interfejs** ima samo jednu metodu. Kao takav, on se lako može konvertovati u lambda rečenicu. Java **Runnable** i **Callable** interfejsi predstavljaju čist primer **funkcionalnih interfejsa**. Za razlikovanje ovih interfejsa od običnih interfejsa, u Javi 8 je dodata nova anotacija *@FunctionalInterface*.

**132.** Šta je default() metoda (kao novina u Javi 8)?

🡪 **Default metode** služe da dodamo nove metode u postojeće interfejse bez da menjamo binarnu kompatibilnost interfejsa sa starim verzijama interfejsa. **Default metode** se smatraju velikom novinom u **Javi 8**.

**133.** Navedite osnovne novine u Javi 9.

🡪 **Moduli sistema Java platforme** predstavljaju novu vrstu komponenti koje se mogu koristiti pri programiranju u Javi. **Modul u Javi** predstavlja kolekciju samo-objašnjivih podataka i programa. U **Javi 9**, daje se uprošćen format za prikazivanje oznake verzije novog izdanja Jave. Takođe, novost jesu i **factory metodi za kolekcije** (**Set**, **List** i **Map**), kojima programski kod postaje dosta lepši i kraći.