Rad sa bazama podataka na programskom jeziku Java

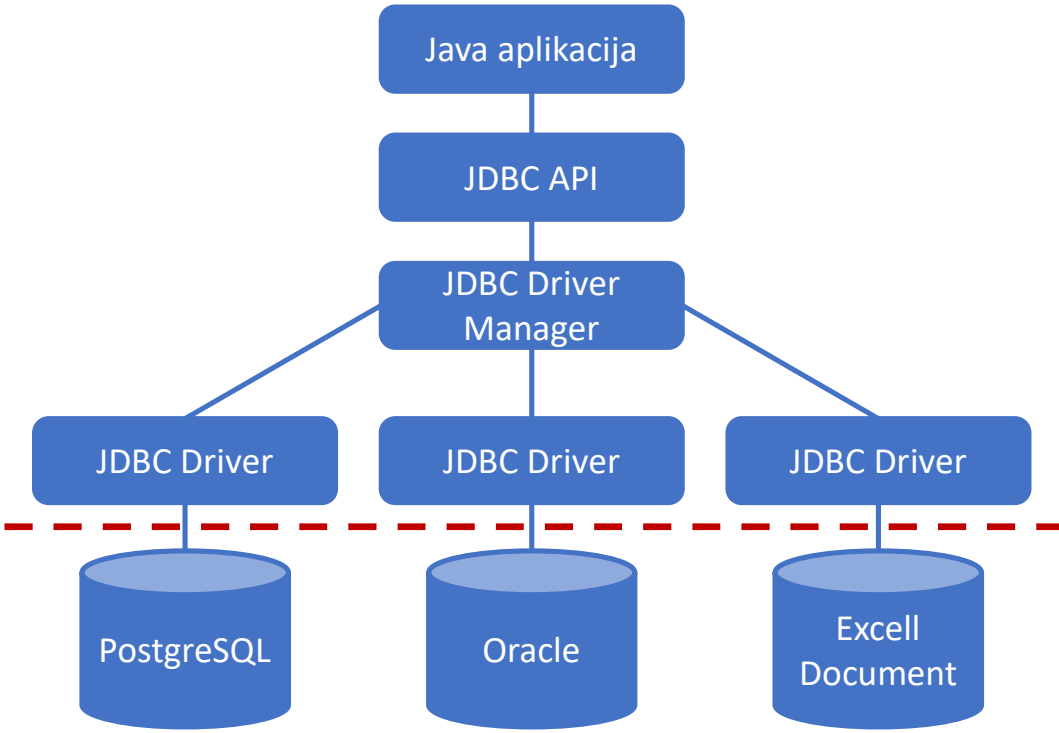
Osnovni koncepti

JDBC API

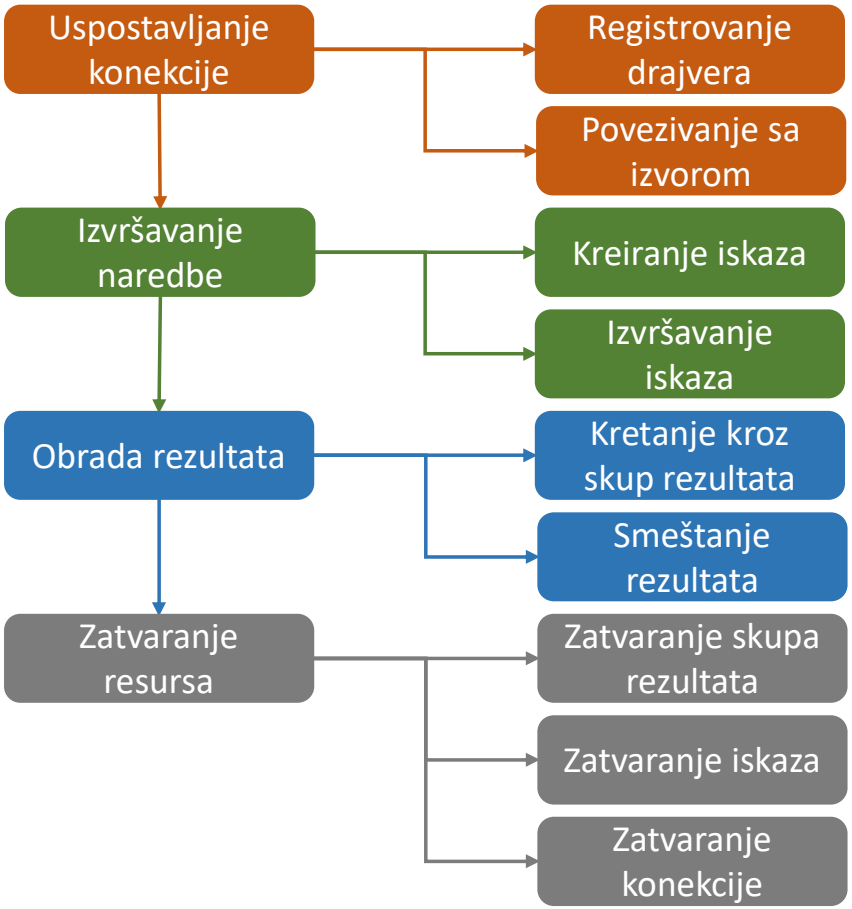
- Java Database Connectivity (JDBC) API
  - Java API (*application programming interface*) za povezivanje programa napisanih na Java programskom jeziku i velikog broja sistema za upravljanje bazama podataka i drugih vrsta izvora podataka.
  - Industrijski **standard** koji obezbeđuje povezivanje koje ne zavisi od korišćenog sistema za upravljanje bazama podataka (*database-independent*).
    - Definisan od strane *Sun Microsystems* (kreatori Java programskog jezika).
    - Dozvoljava proizvođačima SUBP-ova da implementiraju i naslede standard u okviru svojih implementacija **JDBC drajvera**.
  - Celokupan podsistem je definisan u standardnom paketu *java.sql*.
  - Zadaci JDBC-a:
    1. uspostavljanje konekcije sa bazom podataka,
    2. kreiranje i izvršavanje SQL naredbi,
    3. procesiranje rezultata.

JDBC - arhitektura

- **JDBC DriverManager** – komponenta koja upravlja drajverima za različite izvore podataka.
  - Osigurava da se za svaki izvor podataka koristi odgovarajući drajver.
- **JDBC Driver** – komponenta koja omogućava uspostavljanje konekcije sa različitim izvorima podataka.
  - Drajveri implementiraju protokol za prenos podataka između aplikacije i izvora podataka.
  - Za svaki zaseban izvor podataka mora postojati drajver.
- Ovakva arhitektura omogućava razvoj programa nezavisan od pozadinskog izvora podataka (***database-independent***).
  - U slučaju potrebe, moguće je samo zameniti izvor podataka, a da implementacija ostane ista (***hot swapping***).



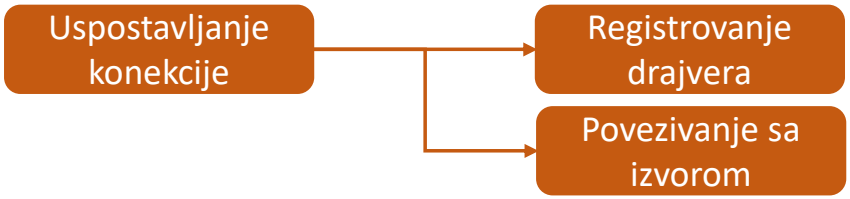
JDBC API – tok izvršavanja naredbe



JDBC - Elementi

- JDBC API elementi pomoću kojih se realizuje izvršavanje naredbi (klase i interfejsi):
  - *DriverManager*
  - *Connection*
  - *Statement*
  - *PreparedStatement*
  - *CallableStatement*
  - *ResultSet*
  - *ResultSetMetaData*
  - *DatabaseMetaData*
  - *SQLException*

JDBC API – uspostavljanje konekcije



JDBC API – uspostavljanje konekcije

- Pozivom sledeće metode kreiramo i registrujemo objekat specificirane drajver klase:

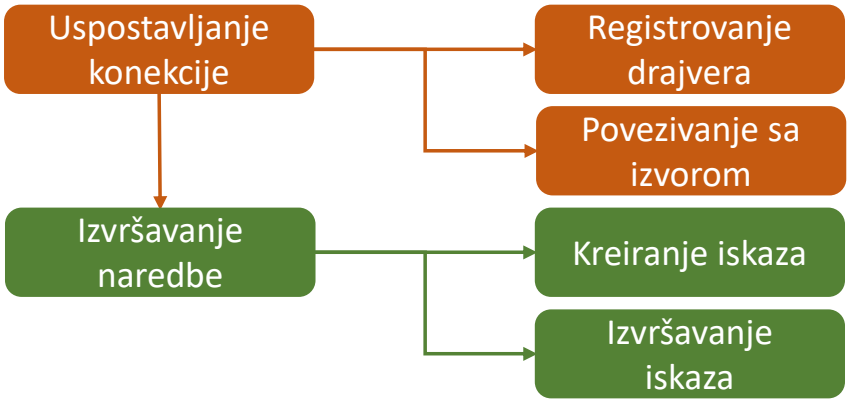
```
Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();
```

- Pozivom sledeće metode specificiramo *DriverManager-u* da pokuša da kreira konekciju na zadati URL, korišćenjem prethodno registrovanog drajvera:

```
Connection connection = DriverManager.getConnection(CONNECTION_STRING,
                                                    USERNAME,
                                                    PASSWORD);
```

  - Pomoću konekcionog stringa specificiramo koji drajver bi trebalo koristiti, kao i parametre konekcije za SUBP koji koristimo:
    - Konekcioni string za rad na lokalnoj mašini: ***"jdbc:oracle:thin:@localhost:1521:xe“***
      - Drajver koji se koristi, IP adresa, port i naziv baze sa kojom se povezujemo
    - Konekcioni string za rad u učionici: ***"jdbc:oracle:thin:@192.168.0.102:1522:db2016"***
  - **Uspostavljene konekcije predstavljaju resurs koji se na kraju rukovanja mora zatvoriti!**

JDBC API – izvršavanja naredbe



JDBC API – izvršavanje naredbe

- Izvršavanje naredbe se sastoji iz dva koraka:
  - kreiranje iskaza:
    - pomoću konekcije kreiraju se objekti klasa koje implementiraju sledeće interfejse:
      - *`Statement`* – koriti se za kreiranje osnovnih iskaza,
      - *`PreparedStatement`* – koristi se za kreiranje parametrizovanih iskaza,
      - *`CallableStatement`* – koristi se za kreiranje iskaza kojima se pozivaju uskladištene procedure i funkcije.
  - izvršavanje iskaza:
    - poziv odgovarajućeg oblika metode za izvršavanje:
      - *`execute()`*,
      - *`executeQuery()`*,
      - *`executeUpdate()`*.

JDBC API – interfejs *Statement*

- Osnovni interfejs za kreiranje i izvršavanje iskaza.
- Objekti klase koja implementira interfejs *Statement* kreiraju se pomoću `connection.createStatement()` metode.
- Može se koristiti za osnovne upite:
  - primeri - *Example01_Query* i *Example02_QueryElegant;*
  - iskazi za upite se izvršavaju pozivom `statement.executeQuery()` metode.
- Ne podržava direktnu parametrizaciju upita:
  - parametrizacija se može izvršiti konkatenacijom stringova;
  - primer – *Example03_QueryWithParams.*
- Može se koristiti i za DDL i DML naredbe:
  - primer – *Example04_DDL_DML_QL;*
  - DDL iskazi se izvršavaju pozivom `statement.execute()` metode;
  - DML iskazi se izvršavaju pozivom `statement.executeUpdate()` metode.

Zadatak za vežbu

- Koristeći JDBC API, napisati program koji će izlistati mbr, ime, prz radnika koji rade na projektu sa šifrom 10, a ne rade na projektu sa šifrom 30.
    - Kreirati novu konekciju koristeći interfejs *`DriverManager`*;
    - Za realizaciju upita koristiti interfejs *`Statement`*.

SQLInjection napad

- *SQLInjection* (*SQLi*) je vrsta injekcionog softverskog napada u okviru kog se maliciozan kod „injektuje“ u string SQL naredbe.
- Kada se ovakva SQL naredba prosledi SUBP-u na parsiranje i izvršavanje, može izazvati nepreviđeno izvršavanje naredbe sa potencijalno katastrofalnim posledicama.
  - Može izazvati curenje „osetljivih“ podataka, modifikaciju podataka, izvršavanje administratorskih naredbi nad bazom podataka, a u nekim slučajevima čak i izdavanje naredbi operativnom sistemu na kom je SUBP pokrenut.
- Jedan od najčešće izvođenih napada!
- Predstavlja zloupotrebu bezbednostih manjkavosti programa – neophodno je iz programa ukloniti bezbednosne propuste.

Interfejs *Statement* i *SQLi* napad

- Konkatenacija stringova radi parametrizacije upita predstavlja veliki potencijalni bezbednosni propust.
  - Podaci koje korisnik treba da unese mogu sadržati maliciozan kod!
  - Primer - *Example05_SQLInjection.*
- Da bi se ovakve situacije izbegle, neophodno je izvršiti „sanaciju“ korisničkog unosa – uklanjanje potencijalno opasnih karaktera iz korisnički unetog teksta.
  - Primer – uklanjanje svih „**‘**“ i „**;**“ karaktera iz korisnički unetog teksta.
- Interfejs *PreparedStatement* podržava automatsku sanitizaciju vrednosti parametara!
- **Kad god se vrši parametrizacija upita, koristiti interfejs *PreparedStatement!***

JDBC API – interfejs *PreparedStatement*

- Izvršavanje upita u okviru većine SUBP-ova podrazumeva prolazak kroz sledeće korake:
  1. parsiranje SQL upita,
  2. kompajliranje SQL upita,
  3. planiranje i optimizaciju načina za dobavljanje rezultata,
  4. izvršavanje optimizovanog upita i vraćanje rezultata.
- Korišćenje interfejsa *`Statement`* podrazumeva da se za svako izvršavanje prolazi kroz sva četiri koraka.
- Korišćenje interfejsa *`PreparedStatement`* podrazumeva jedan prolazak.
  - Kroz prva tri koraka – prilikom kreiranja objekta klase koja implementira interfejs *PreparedStatement*
  - **Svako izvršavanje naredbe se svodi na izvršavanje 4. koraka**, uz opciono prosleđivanje vrednosti za parametre.
  - Značajno efikasnije prilikom ponovnog korišćenja istog objekta – **vrlo pogodno za parametrizaciju**.

JDBC API – interfejs *PreparedStatement*

- Specificiranje parametara:
  - mesto u upitu na koje bi trebalo da dođe konkretna vrednost označava se karakterom „**?**“.
- Postavljanje vrednosti parametara:
  - za postavljanje konkretne vrednosti na mesto parametra koriste se metode oblika *`setX(int parameterIndex, X value)`*:
    - *parameterIndex* – indeks parametra u upitu:
      - zavisi od redosleda karaktera „**?**“ u upitu;
      - **indeksiranje kreće od 1**;
    - *X* – predstavlja tip podatka za koji se postavlja vrednost;
    - *value* – vrednost koja se postavlja za parametar.
  - Često korišćene metode:
    - *`setString`*`(...)`,
    - *`setInt`*`(...)`,
    - *`setFloat`*`(...)`,
    - *`setDate`*`(...)`,
    - *`setBoolean`*`(...)`,
    - *`setNull`*`(...)...`

JDBC API – interfejs *PreparedStatement*

- Objekti klase koja implementira interfejs *`PreparedStatement`* kreiraju se pomoću *`connection.prepareStatement(String statement)`* metode.
- Može se koristiti za osnovne upite:
  - primer: - *Example01_Query;*
  - iskazi za upite se izvršavaju pozivom *`.executeQuery()`* metode.
- Parametrizacija je **direktno podržana**:
  - neophodno izvršiti specificiranje parametara prilikom kreiranja naredbe;
  - neophodno izvršiti postavljanje vrednosti parametara pre izvršavanja naredbe;
  - primeri – *Example02_QueryWithParams, Example03_SQLInjection.*
- Može se koristiti i za DDL i DML naredbe:
  - primer – *Example04_DDL_DML_QL;*
  - DDL iskazi se izvršavaju pozivom *`.execute()`* metode;
  - DML iskazi se izvršavaju pozivom *`.executeUpdate()`* metode.

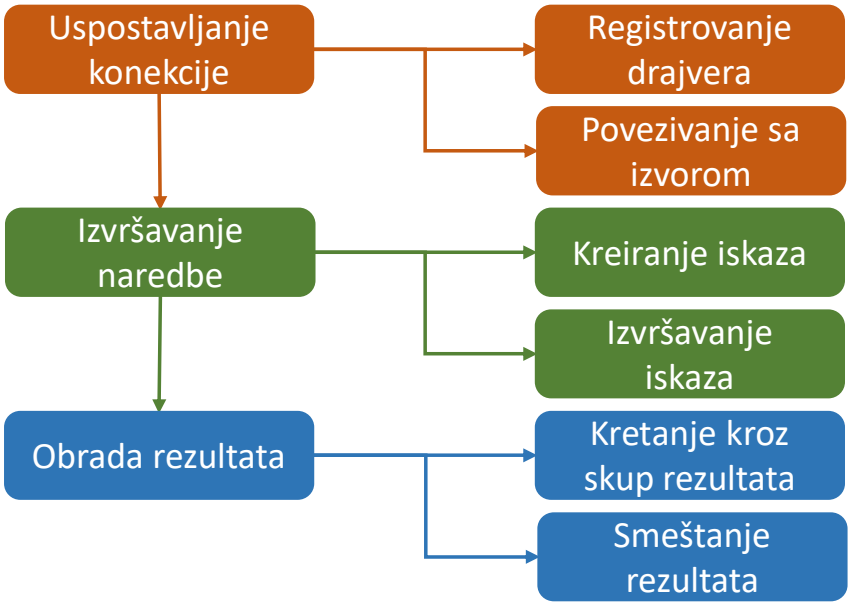
Zadatak za vežbu

- Koristeći JDBC API, napisati program koji će za unete spr i brc vrednosti izlistati broj radnika koji na zadatom projektu rade više od zadatog broja sati.
    - Za realizaciju upita koristiti interfejs *`PreparedStatement`*.

JDBC API – *Callable statement*

- Objekti klase koja implementira interfejs *`CallableStatement`* koriste se za izvršavanje uskladištenih procedura i funkcija.
- Kreiraju se pomoću *`connection.prepareCall(String statement)`* metode.
  - Primer: Example01_ExecuteFunction
    - Pre izvršavanja neophodno kreirati odgovarajuću uskladištenu funkciju:
      - *Example01_SelectProjekatFunction.sql;*
    - iskazi za upite se izvršavaju pozivom *`.execute()`* metode, dok se vrednosti izlaznih parametara preuzimaju pozivom *`.getObject(int parameterIndex)`*.
- Registrovanje izlaznih parametara:
  - Za registrovanje izlaznih parametara, preko kojih će biti vraćena vrednost iz procedure ili funkcije, koriste se metode oblika:
    - *`registerOutParameter(int parameterIndex, int sqlType)`*:
      - *parameterIndex* – indeks parametra u upitu:
        - zavisi od redosleda karaktera „**?**“ u upitu;
        - indeksiranje kreće od 1;
      - *sqlType* – predstavlja tip parametra.

JDBC API – obrada rezultata



Kursori

- Kursori predstavljaju kontrolnu strukturu pomoću koje je moguće vršiti prolazak kroz skup rezultata (***result set***) nekog upita po principu red-po-red.
  - Umesto da se ceo upit izvrši odjednom, kreira se kursor koji omogućava postupno čitanje rezultata upita.
- Jedan od razloga za ovakvo rukovanje podacima jeste izbegavanje situacije u kojoj je odjednom potrebno obraditi veliku količinu memorije, što može izazvati nedostatak memorije za obradu podataka (***memory overrun***).
- Predstavljaju koncept vrlo sličan konceptu iteratora u različitim programskim jezicima.
  - Kursor pokazuje na red koji je trenutno na redu za obradu.

JDBC API – interfejs *ResultSet*

- Koristi se za rukovanje pozadinskim kursorskim kontrolnim strukturama – kako bi se izvršila obrada skupa rezultata dobijenog izvršavanjem zadatog upita.
- Objekti klase koja implementira interfejs *ResultSet* kreiraju se pozivom *.executeQuery()* naredbe nad objektima iskaza.
- Primer iteriranja kroz rezultat upita - *Example01_Iterating*
  - Neposredno nakon kreiranja objekta, kursor pokazuje na poziciju pre prvog reda (*beforeFirst*).
  - Tip se definiše prilikom kreiranja objekta iskaza, prosleđivanjem odgovarajuće konstante. Tipovi kursora:
    - TYPE_FORWARD_ONLY – koriste se samo za kretanje u napred, podrazumevani tip;
    - TYPE_SCROLL_INSENSITIVE – *scrollable* kursor koji nije osetljiv na promene u originalnom skupu rezultata;
    - TYPE_SCROLL_SENSITIVE – *scrollable* kursor koji je osetljiv na promene u originalnom skupu rezultata.

Zadatak za vežbu

- Za svakog šefa ispisati ime, prezime, broj radnika kojima je direktno nadređeni i platu. Takođe, ispod svakog šefa dati prikaz svih radnika (ime, prezime i plata) kojima je direktno nadređeni.
  - Primer izlaza programa:

```
Savo Oroz,  2,  40000din
    Pero Peric, 30000din
    Eva  Ras,   40000din
...
```

JDBC API – interfejs *ResultSet*

- Preuzimanje vrednosti atributa:
  - Za preuzimanje vrednosti atributa trenutno obrađivane torke koriste se metode oblika *.getX(int columnIndex)*.
    - *X* – predstavlja tip podatka za koji se postavlja vrednost;
    - columnIndex – redni broj kolone rezultujućeg upita; indeksiranje počinje od 1.
  - Često korišćene metode:
    - `getString(...)`,
    - `getInt(...)`,
    - `getFloat(...)`,
    - `getDate(...)`,
    - `getBoolean(...)...`
  - Preuzete vrednosti se dalje mogu dodeliti nativnim java tipovima, objektima korisnički kreiranih klasa...

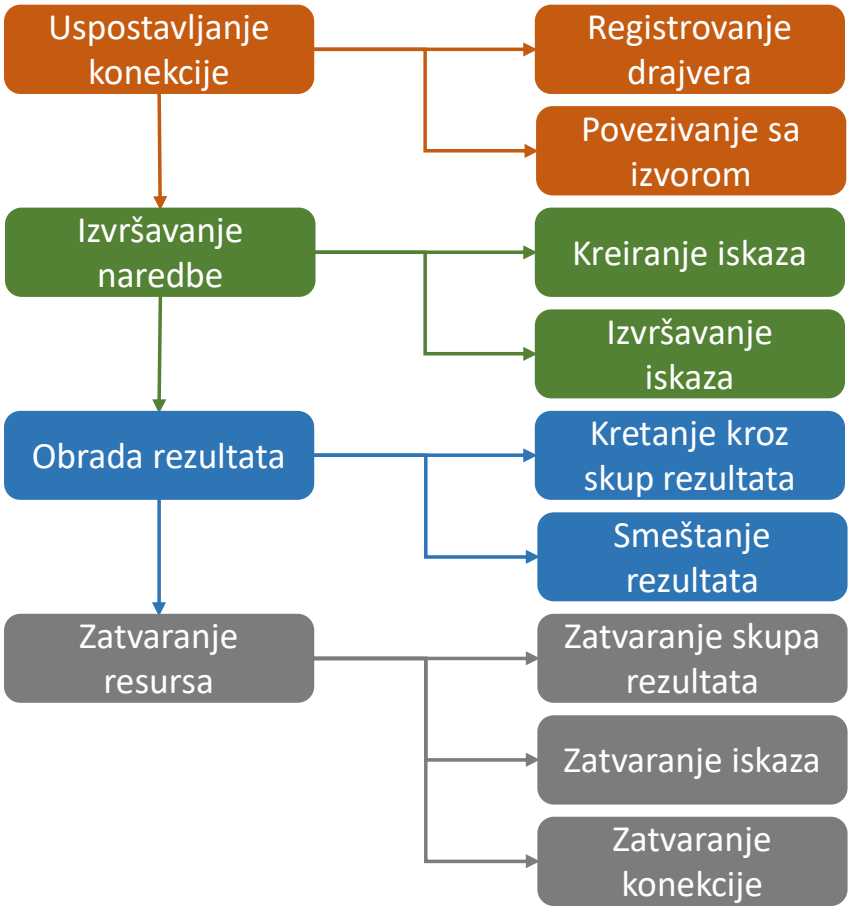
JDBC API – interfejs *ResultSet*

- Interfejs ResultSet može se koristiti i za ažuriranje podataka prilikom prolaska kroz rezultujuči skup podataka.
  - Definiše se prilikom kreiranja objekta iskaza, prosleđivanjem konstante CONCUR_*UPDATABLE*.
    - Može se definisati i da ažuriranje nije dozvoljeno, korišćenjem konstante *CONCUR_READ_ONLY* – podrazumevano ponašanje.
- Primer ažuriranja podataka – *Example02_Updating*
  - Unos:
    - podrazumeva pozicioniranje na specijalni red pod nazivom *InsertRow;*
    - potom se vrši postavljanje vrednosti za kolone nove torke korišćenjem metoda oblika `updateX(int columnIndex, X value);`
    - kada se završi postavljanje vrednosti, unos se vrši pozivom `insertRow()` metode;
    - **uneta torka NEĆE biti vidljiva u originalnom skupu rezultata!**
  - Izmena
    - podrazumeva pozicioniranje na žejeni red;
    - potom se vrši postavljanje novih vrednosti za kolone torke korišćenjem metoda oblika `updateX(int columnIndex, X value);`
    - kada se završi postavljanje vrednosti, unos se vrši pozivom `updateRow()` metode;
    - **unete izmene će biti vidljive u originalnom skupu rezultata.**
  - Brisanje
    - podrazumeva pozicioniranje na željeni red;
    - pa potom poziv metode `deleteRow();`
    - **torka će biti obrisana i iz originalnog skupa torki.**

Zadatak za vežbu

- U tabelu radnik dodati novog radnika, a potom ga angažovati na projektu sa najmanjom vrednošću obeležja spr.
    - Za realizaciju unosa koristiti interfejs *`ResultSet`.*

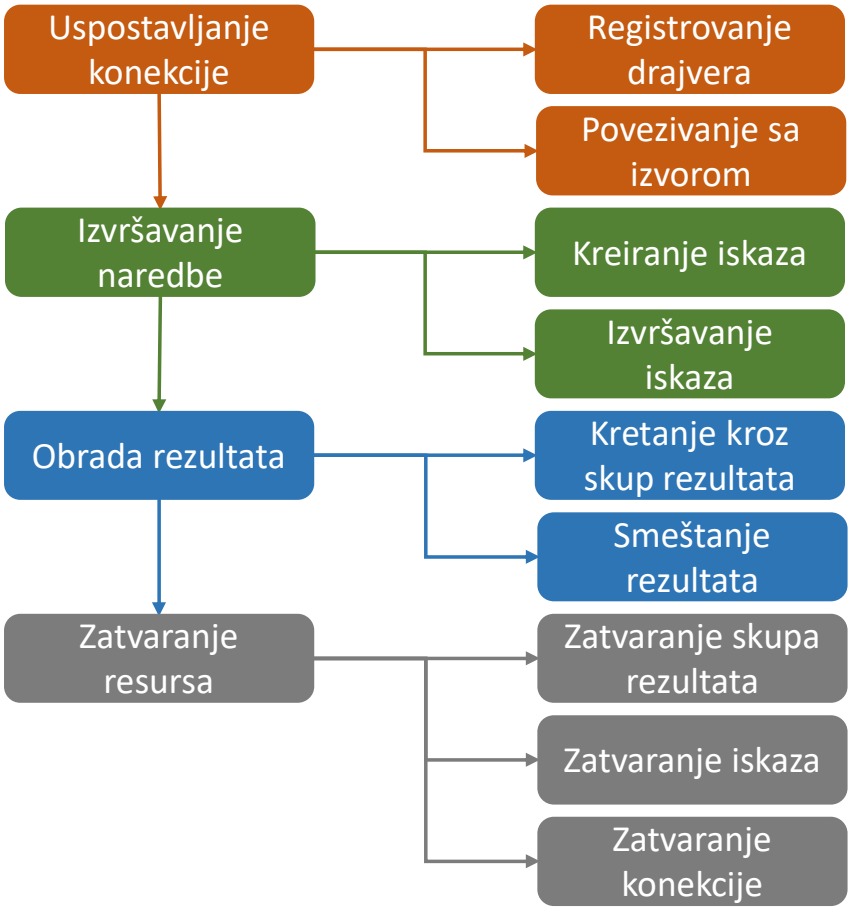
JDBC API – zatvaranje resursa



JDBC API – zatvaranje resursa

- Resursi koji se koriste prilikom rada sa različitim izvorima podataka obavezno se moraju eksplicitno zatvoriti.
  - Da bi se oslobodili zauzeti resursi – kursorska područja, *socket*-i za konekciju itd.
  - Na ovaj način se izbegava „curenje“ memorije (eng. *memory leak*).
- Resursi koji se moraju zatvoriti: `Connection`, `Statement`, `PreparedStatement`, `CallableStatement` i `ResultSet`.
- Zatvaranje resursa ne može se prepustiti *garbadge collector*-u!
  - Nije u mogućnosti da zatvori kursorsko područje u okviru SUBP-a.
- Za eksplicitno zatvaranje resursa koristiti:
  - *try-with-resource* iskaz za automatsko zatvaranje resursa, ili
  - eksplicitni poziv `.close()` metode nad resursom,
    - najčešće u sklopu *finally* bloka običnog *try-catch* bloka.

JDBC API – zatvaranje resursa



Napredne opcije

JDBC API – transakcije

- JDBC API pruža mogućnost upravljanja transakcijama korišćenjem objekata konekcije.
- Podrazumevani režim rada je *auto commit*, u kom se svaka izvršena naredba tretira kao zasebna transakcija – nakon svake uspešno izvršene naredbe pozvaće se naredba *commit*.
  - Ovakvo ponašanje može biti problematično u situaciji kada se vrši kompleksnija obrada podataka – integritet podataka može biti doveden u pitanje.
  - Primer - *Example01_AutoCommit*.
- Kako bi se ručno upravljalo transakcijama, neophodno je isključiti *auto commit* režim pozivom metode `setAutoCommit(false)` nad objektom konekcije koji koristimo za kreiranje iskaza.
  - Potom je moguće u odgovarajućem trenutku pozvati *commit* ili *rollback* naredbu, takođe korišćenjem objekta konekcije.
  - Primer - *Example02_ManualCommit.*

JDBC API – meta-podaci

- U nekim slučajevima može biti neophodno da se pristupi podacima o samoj bazi podataka i o tabelama u njoj – rečniku podataka (*data dictionary*).
- U tu svrhu postoje dve interfejsa:
  - `DatabaseMetaData`,
  - `ResultSetMetaData`.
- Primeri upotrebe:
  - `DatabaseMetaData.getTables()`,
  - `ResultSetMetaData.getColumnName()`,
  - `ResultSetMetaData.getColumnTypeName()`.

Dobre prakse

Connection Pooling

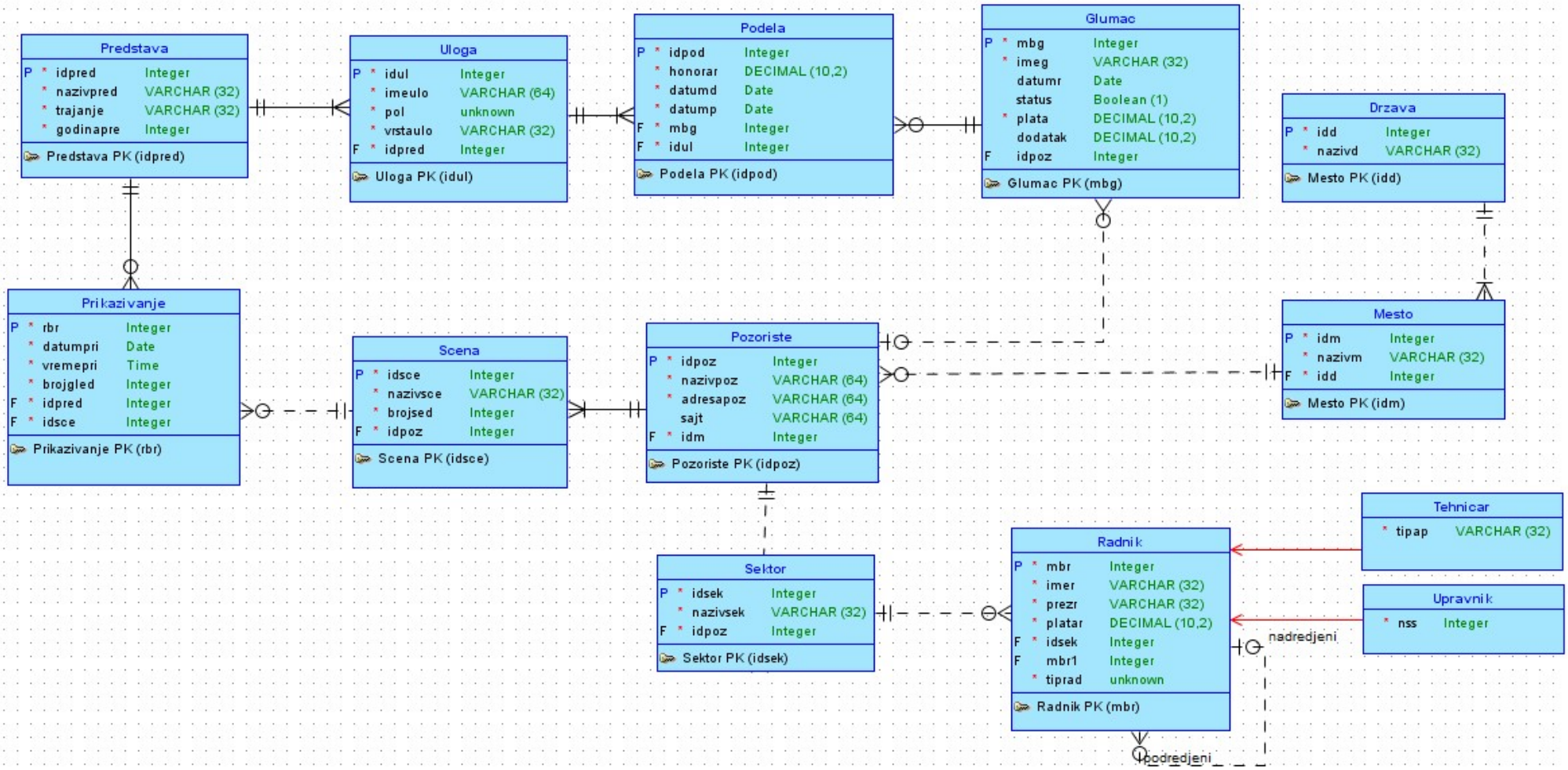
- Svako uspostavljanje veze sa izvorom podataka zahteva zauzimanje resursa za održavanje konekcije, kao i inicijalnu komunikaciju sa izvorom podataka.
- U aplikacijama koje intenzivno koriste usluge izvora podataka, ovo može imati značajan uticaj na performanse, jer se postupak kreiranja stalno iznova ponavlja.
- Zbog toga je uveden *Connection Pooling* koncept, koji predviđa keširanje određenog broja unapred pripremljenih konekcija.
  - Kad god je potrebna konekcija radi komunikacije sa izvorom podataka, preuzmemo je iz unapred pripremljenog „bazena“ konekcija.
  - Kada se završi rad sa konekcijom, ona se vraća u „bazen“ pozivom *.close()* metode.
- Jedna od najkorišćenijih Java implementacija koncepta – *HikariCP*.
  - Primer - *ConnectionUtil_HikariCP* i *Example01_ConnectionPool*.

DAO šablon

- *Data Access Object* (*DAO*) šablon je strukturalni šablon koji omogućava razdvajanje sloja poslovne logike (*bussines layer*) od sloja za perzistenciju podataka (*persistence layer*).
- Ovakvim razdvajanjem slojeva održava se princip jedinstvenog zaduženja svake komponente sistema (*Single Responsibility*)
  - Što je u skladu sa *SOLID* dizajn principima softvera, čiji je cilj da softversko rešenje bude razumljivo, fleksibilno i održivo.
- DAO sloj implementira logiku nad konkretnim entitetima – sadrži metode poput *getX*, *insert*, *update*, *delete*...
  - Za svaki entitet postojaće i zasebna klasa u DAO sloju.
  - Ostatak aplikacije komunicira sa izvorom podataka isključivo preko metoda koje pruža ovaj sloj.

Primer realne aplikacije

Šema baze podataka



Arhitektura aplikacije

- Razdvajanjem slojeva održava se princip jedinstvenog zaduženja svake komponente sistema čime se obezbeđuje fleksibilno i održivo rešenje:
  - Model
  - DTO - klase iz paketa DTO (*Data Transfer Object*) se koriste za koriste za kreiranje objekata koji preuzimaju kombinovane podatke iz različitih tabela i prosleđuju se sloju biznis logike na obradu
  - DAO -
  - *Service*
  - *User Interface Handler*

Model

- Primer : klase iz paketa model
- U ovom sloju se kreiraju klase koje su neophodne za mapiranje podataka koji se preuzimaju iz baze podataka na objekte kojima se manipuliše u aplikaciji.
- Svaka tabela iz baze podataka ima svoju odgovarajuću klasu u modelu.
- Klase iz modela sadrže: polja koja odgovaraju po tipu kolonama iz baze podataka, konstukror, *get*() i *set*() metode, predefinisane metode ispisa…

DTO (*Data Transfer Object*)

- Primer : klase iz paketa dto
- *Data Transfer Object* je sloj koji služi za prihvatanje podataka iz baze i prosleđivanje ovih objekata sloju koji vrši biznis logiku.
- Prilikom kompleksnih upita može se desiti da upit vraća podatke dobavljene iz različitih tabela (klase modela nisu dovoljne za preuzimanje takvih podataka jer tu važi mapiranje jedna tabela jedna klasa modela).
- Atributi DTO klase služe za mapiranje kolona iz različitih tabela.

DAO

- Primer: klase iz paketa dao
- *Data Access Object* (*DAO*) šablon je strukturalni šablon koji omogućava razdvajanje sloja poslovne logike (*bussines layer*) od sloja za perzistenciju podataka (*persistence layer*).
- Ovakvim razdvajanjem slojeva održava se princip jedinstvenog zaduženja svake komponente sistema (*Single Responsibility*)
  - Što je u skladu sa *SOLID* dizajn principima softvera, čiji je cilj da softversko rešenje bude razumljivo, fleksibilno i održivo.
- DAO sloj implementira logiku nad konkretnim entitetima – sadrži metode poput `getX`, `insert`, `update`, `delete`...
  - Za svaki entitet postojaće i zasebna klasa u DAO sloju.
  - Ostatak aplikacije komunicira sa izvorom podataka isključivo preko metoda koje pruža ovaj sloj.

CRUDDao interfejs

- Služi kako bi se postiglo generičko rešenje za implementaciju osnovnih *CRUD* (*create, read, update, delete*) operacija:

```
public interface CRUDDao<T, ID> {

    int count() throws SQLException;

    void delete(T entity) throws SQLException;

    void deleteAll() throws SQLException;

    void deleteById(ID id) throws SQLException;

    boolean existsById(ID id) throws SQLException;

    Iterable<T> findAll() throws SQLException;

    Iterable<T> findAllById(Iterable<ID> ids) throws SQLException;

    T findById(ID id) throws SQLException;

    void save(T entity) throws SQLException;

    void saveAll(Iterable<T> entities) throws SQLException;
}
```

Service

- Primer: klase iz paketa service
- Servis predstavlja međusloj između elemenata korisničkog interfejsa i DAO sloja.
- U ovom sloju implementirane su metode koje implementiraju logiku kompleksnijih izveštaja.

User Interface Handler

- Primer: klase iz paketa ui_handler
- Klase koje služe za kreiranje ispisa na konzoli i prihvatanje korisničkog unosa.

Zadaci

- Za svako pozorište prikazati listu scena koje ima. Ukoliko pozorište nema scenu ispisati: NEMA SCENE

- **Model** : klase `Pozoriste, Scena`
- **UIHandler:** klasa `ComplexUIHandler`
- **Service**: klasa `ComplexFuncionalityService`, *metoda* *`showSceneForTheatre()`*
- **DAO**: klasa `PozoristeDAO`, metoda *`findAll()`*
  klasa `ScenaDAO`, metoda *`findSceneByTheatre()`*

Zadaci

- Prikazati informacije o predstavama koje se prikazuju. Pored osnovnih informacija o predstavama prikazati sva prikazivanja za svaku od njih. Za svako pozorište prikazati ukupan broj gledalaca, prosečan broj gledalaca i broj prikazivanja.

- **Model** : klase `Predstava, Prikazivanje`
- **UIHandler:** klasa `ComplexUIHandler`
- **DTO:** klasa `PrikazivanjeDTO`
- **Service**: klasa `ComplexFuncionalityService`, *metoda* `showReportingForShowingShows()`
- **DAO**: klasa `PrikazivanjeDAO`, metoda `findSumAvgCountForShowingShow(), findAllDistinctShowFromShowing(), findShowingByShowId()`
  klasa `PredstavaDAO`, metoda *`findById()`*

Zadaci

- Prikazati nazive scena i broj sedišta za sve scene čiji broj sedišta je u intervalu plus/minus 20% od broja sedišta koje ima scena Scena Joakim Vujic pozorišta Knjazevsko-srpski teatar Kragujevac. Za sve predstave koje se prikazuju na tim scenama izračunati ukupan broj gledalaca. Prikazati samo one predstave čiji je ukupan broj gledalaca veći od 700. Za te predstave prikazati ukupan broj uloga na toj predstavi. Za scene na kojima se ne prikazuju predstave ispisati poruku.

- **Model** : klase `Predstava, Scena, Prikazivanje`
- **UIHandler:** klasa `ComplexUIHandler`
- **DTO**: klasa `PrikazivanjeScenaDTO`
- **Service**: klasa `ComplexFuncionalityService`, *metoda* *`showComplexQuery()`*
- **DAO**: klasa `PredstavaDAO`, metoda *`findById(), findCountOfRoles()`*
  klasa `ScenaDAO`, metoda *`findSceneForSpecificNumberOfSeats ()`*
  klasa `PrikazivanjeDAO`, metoda *`findBySceneId()`*

Zadaci

- Prikazati id, nazive i prosecan broj gledalaca predstava koje imaju najveci prosecan broj gledalaca. Za te predstave prikazati listu uloga. Pored toga prikazati koliko ukupno ima muških uloga i koliko ukupno ima ženskih uloga.

- **Model** : klase `Predstava, Uloga`
- **UIHandler:** klasa `ComplexUIHandler`
- **Service**: klasa `ComplexFuncionalityService, metoda showMostVisitedShow()`
- **DAO**: klasa `PredstavaDAO`, metoda `findMostVisitedShows ()`
  klasa `UlogaDAO`, metoda `findRoleByTheatreId (), findCountForRoleGender(), findCountForRoleGender()`

Zadaci – Samostalni rad

- Za podelu napraviti CRUD operacije i kroz aplikaciju uneti nekoliko torki u tabelu Podela. Nakon toga napraviti izveštaj koji će za svakog glumca prikazati ime glumca, naziv predstave, uloge u predstavi i honorar za koju je dobio najveći honorar. Za glumce koji nemaju podelu ispisati umesto naziva predstave i uloge NEMA, a umesto honorara staviti 0.

Zadaci – Samostalni rad

- Prikazati sve uloge koje nisu podeljene. Zatim za te uloge prikazati spisak glumaca koji rade u pozorištu u kome se ta predstava prikazuje u nekom narednom periodu, a koji nemaju nijedan angažman.

Zadaci – Samostalni rad

- Prikazati za svakog glumca kojem je dodeljena neka uloga, mbg, prezg, imeg, imepred, imeulo i spisak drugih glumaca kojima je dodeljena ista uloga. Pored toga neophodno je prikazati ukupan broj drugih glumaca kojima je dodeljena ista uloga.

Zadaci – Samostalni rad

- Prikazati za svakog glumca kom je dodeljena uloga, mbg, prezg, imeg, kao i imepred, imeulo prikazati spisak glumaca za dodeljenu ulogu. Pored toga prikazati udeo tog glumca u ukupnom honoraru koji se izdvaja za dodeljenu ulogu za sve njene glumce. Udeo izraziti u procentima zaokruženo na dve decimale.

Zadaci – Samostalni rad

- Prikazati matični broj, ime, prezime i platu glumca i listu ostalih glumaca i njihovih honorara za tu ulogu. Prikazati samo one glumce čiji je honorar za neku ulogu veći od prosečnog honorara za tu ulogu.

Zadaci – Samostalni rad

- Za svakog glumca prikazati koliki je ukupni honorar glumaca na svim njegovim predstavama. Uzeti u obzir samo glumce koji glume i u predstavama koje se ne prikazuju u njihovom matičnom pozorištu.

Zadaci

- Prikazati prikazivanja predstava u narednom periodu na scenama na kojima je broj gledalaca veći od broja sedišta na toj sceni. Obrisati te torke. Raspodeliti sedišta tako da se uvedu novi termini prikazivanja ove predstave na toj sceni. Zauzeti pune scene onoliko puta koliko je potrebno i napuniti poslednju scenu sa onoliko mesta koliko je ostalo. Za novi datum uneti današnji datum. Ispisati sva prikazivanja.

- **Model** : klase `Prikazivanje, Scena`
- **UIHandler:** klasa `ComplexUIHandler`
- **DTO**: klasa `PrikazivanjeDeleteDTO`
- **Service**: klasa `ComplexFuncionalityService`, *metoda* `showShowingForDeleting ()`
- **DAO**: klasa `PrikazivanjeDAO`, metoda *`findShowingForDeleting(), findAll()`*

Zadaci – Samostalni rad

- Omogućiti interaktivni unos novog pozorišta za koji je potrebno uneti sve osnovne podatke, scene i mesto pozorišta. Ukoliko mesto ne postoji dodati ga u bazu podataka, a ukoliko postoji samo povezati. Scene kreirati u bazi podataka.

Zadaci - Samostalni rad

- Rebalansirati opterećenje glumcima starijim od 60 godina tako da u jednoj nedelji mogu da glume u najviše dve predstave. Sve uloge koje ostanu slobodne rebalansiranjem ovakvih glumaca, dodeliti drugim glumcima koji su bili prethodno u podeli za tu ulogu. Ako ne postoje takvi glumci prikazati uloge koje su ostale neupražnjene. Ukoliko novom podelom postoje prikazivanja predstava koje se daju u istom terminu sa istim glumcem u različitim ulogama, premestiti jedno od prikazivanja u drugi termin.

Kraj prezentacije