
Razvoj programskih rješenja

Grafički korisnički interfejs

Grafički korisnički interfejs (GUI)

- Programi koje smo do sada pravili koriste *standardni ulaz i izlaz* (unos podataka sa tastature i prikaz u formi teksta). Mnoge sistemske i serverske aplikacije se i danas razvijaju na taj način.
- Ipak, danas su korisnici navikli da imaju na raspolaganju *grafičko okruženje* u kojem mogu kontrolisati programe izborom postojećih komponenti (meniji, dugmad itd.)
- Suštinska razlika između komandno-linijskih i grafičkih aplikacija je da u GUI aplikacijama main funkcija postaje nebitna, te najčešće sadrži samo startni kod za ostatak programa.

Programiranje vođeno događajima

- *Programiranje vođeno događajima* (eng. event-driven programming) je *programska paradigma* kod koje se programi ne izvršavaju sekvencijalno, nego je izvršenje određenih procedura (funkcija) uzrokovano nekim događajima izvana. Npr. kada pokrenete GUI program, on prikazuje ekran i zatim čeka na vaše akcije
- Osnovni element GUI programa je *petlja događaja* (eng. event loop) koja je u suštini beskonačna petlja u kojoj se provjerava da li je došlo do događaja. Ako se desi događaj, poziva se odgovarajuća funkcija *rukovaalac događaja* (eng. event handler)

Programiranje vođeno događajima (2)

- **Primjer:** recimo da trebamo napraviti klasu Alarm koja reaguje na dva događaja: ako je vrijeme za alarm, treba početi zvoniti, a ako korisnik pritisne tipku Reset treba prekinuti zvono.

```
class Alarm {  
    Bell bell = new Bell();  
    public void handleTimeout() {  
        bell.startRinging();  
    }  
    public void handleReset() {  
        bell.stopRinging();  
    }  
}
```

Programiranje vođeno događajima (3)

- Potrebno je da u klasi Alarm imamo metodu sa petljom događaja:

```
void eventLoop() {  
    while(true) {  
        if (System.currentTimeMillis() == alarmTime) {  
            handleTimeout();  
        }  
        if (resetButton.isPressed())  
            handleReset();  
        }  
        Thread.sleep(100);  
    }  
}
```

Programiranje vođeno događajima (4)

- Funkcija **eventLoop** može biti i **main()** funkcija
- Beskonačne petlje opterećuju procesor računara i usporavaju ostale programe koji se izvršavaju! Zbog toga je potrebno omogućiti procesoru "da diše". To smo postigli naredbom:

```
Thread.sleep(100)
```

- Ova naredba pauzira izvršenje petlje 100 ms, a za to vrijeme procesor nije opterećen i mogu se izvršavati drugi programi

Biblioteke za grafičko okruženje

Razvoj biblioteka za grafičko programiranje možemo pratiti po generacijama:

- 1. Prva generacija** (80te i 90te godine) - GUI toolkits - biblioteke koje su nudile funkcije za crtanje osnovnih (primitivnih) oblika: prozor, dugme, upravljanje mišem itd. ali je programer sam morao praviti petlju događaja ili pozvati default petlju. Primjeri: Win32, MFC, wxWidgets, Gtk.
- 2. Druga generacija** (00te godine) - GUI frameworks - koriste inverziju kontrole (IoC) tako da vaš main postaje samo poziv "master funkcije" frameworka, a dalje se sve postiže konfiguracijom, nasljeđivanjem klasa i slično. Primjeri: Windows Forms, Qt,
- 3. Treća generacija** (10te godine) - deklarativni interfejs (eng. declarative UI) - poseban jezik (najčešće XML) se koristi da se opiše kako treba izgledati grafički interfejs, a kod se piše samo za event handlera i to samo ako default handleri nisu dovoljni. Primjeri: WPF, UWP.

Dizajneri korisničkog interfejsa

- Dizajner ili builder (graditelj) korisničkog interfejsa je program u kojem se može nacrtati kako želite da izgleda korisnički interfejs vašeg programa, a on će izgenerisati potrebni kod
- Sa bibliotekama prve i druge generacije generiše se programski kod koji je vrlo težak za ručnu promjenu, tako da se većinom ne preporučuje da ga mijenjate osim kroz UI dizajner
- Kod treće generacije generiše se XML koji je lakši za održavanje i manja je vjerovatnoća grešaka

Java UI

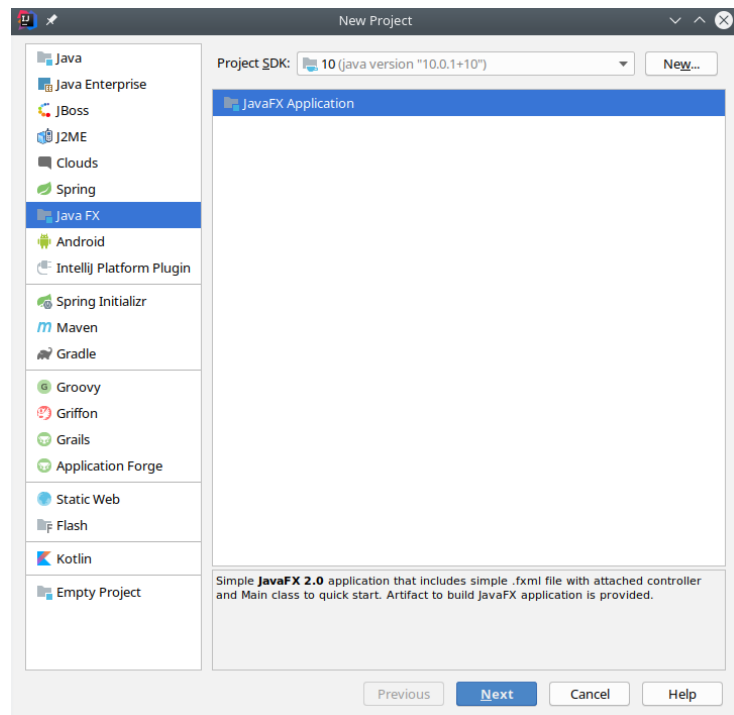
- Jedan od prvobitnih ciljeva Java programskog jezika je bio olakšan razvoj GUI aplikacija. Od prve verzije Java uključuje biblioteku za razvoj UI
- **AWT** biblioteka dostupna od početka (sredina 90tih), vrlo zastario i primitivan izgled čak i za to vrijeme! Biblioteka prve generacije
- **Swing** uključena od Java 1.2, nadograđuje funkcionalnosti AWT, omogućava moderan korisnički interfejs, i danas je "default" grafičko okruženje mada se više ne razvija aktivno

Java UI (2)

- **SWT** je dio Eclipse projekta, adresira nedostatke Swing-a: sporost i izgled koji nije potpuno identičan nativnim aplikacijama. Kompleksna instalacija i korištenje (van Eclipse okruženja), nije zaživio
- **JavaFX** biblioteka treće generacije prvobitno namijenjena za web i mobilne aplikacije, danas je popularna i na desktop računarima. Od Java 8 je uključena u SDK ali nije zvanično dio Java specifikacije (za razliku od Swing-a). JavaFX 2 je posljednja verzija koja podržava Java 7, ali je vrlo zastarjela i ne preporučuje se njeno korištenje
- *Radićemo JavaFX framework, mada se Swing još uvijek više koristi*

Kreiranje JavaFX projekta

- Kreirajte novi projekat kroz IntelliJ IDEA, izaberite tip projekta JavaFX Application
- Obratite pažnju da JavaFX ne dolazi uz neke verzije JDK i mora se posebno downloadovati. Od verzije 11 JavaFX više ne dolazi uz Oracle JDK niti OpenJDK, nego se mora posebno downloadovati sa stranice openjfx.io.



JavaFX Hello World

```
public class Main extends Application {  
    @Override  
    public void start(Stage primaryStage) throws Exception{  
        Parent root = FXMLLoader.load(getClass().getResource("sample.fxml"));  
        primaryStage.setTitle("Hello, World!");  
        primaryStage.setScene(new Scene(root, 300, 275));  
        primaryStage.show();  
    }  
  
    public static void main(String[] args) {  
        launch(args);  
    }  
}
```

JavaFX Hello World

- Biće kreiran Hello World projekat koji na ekranu prikazuje prazan prozor.

Analizirajmo ovaj program:

- Klasa **Main** nasljeđuje JavaFX klasu **Application**:
`public class Main extends Application {`
- Metoda **main** samo poziva **launch** metodu Application klase (inverzija kontrole):

```
public static void main(String[] args) { launch(args); }
```

Ova main metoda nije ni potrebna ako pokrećete projekat iz komandne linije

- Nova metoda **start** nam služi da pokrenemo kreiranje raznih objekata koji su nam potrebni za rad:

```
public void start(Stage primaryStage) throws Exception { ... } 13
```

Osnovni pojmovi JavaFX biblioteke

- JavaFX koristi pozorišnu terminologiju:
 - Primarna klasa unutar koje se kreiraju svi elementi vizualnog okruženja zove se **Stage** (pozornica). **Stage** objekat jednostavno rečeno je prozor koji će biti kreiran
 - Sadržaj prozora nalazi se u klasi **Scene**. Nakon kreiranja **Stage** i postavljanja osnovnih parametara, kreira se **Scene** objekat koji se postavlja na **Stage**. **Stage** će se prilagoditi dimenzijama **Scene** objekta
 - Unutar **Scene** zatim možete postavljati klase koje odgovaraju drugim kontrolama

Primjer - kreiranje UI iz

Nepoznata klasa Button
rješava se pritiskom na
Alt+Enter

```
public void start(Stage primaryStage) throws Exception {  
    Scene scene = new Scene(new Button("OK"), 200, 250);  
    primaryStage.setTitle("MyJavaFX"); // Set the stage title  
    primaryStage.setScene(scene); // Place the scene in the stage  
    primaryStage.show(); // Display the stage  
    Stage stage = new Stage(); // Create a new stage  
    stage.setTitle("Second Stage"); // Set the stage title  
    // Set a scene with a button in the stage  
    stage.setScene(new Scene(new Button("New Stage"), 100, 100));  
    stage.show(); // Display the stage  
}
```

Layout panes

- Uočavamo da je dugme razvučeno preko čitavog prozora, što nije pretjerano lijepo. To će se desiti sa bilo kojom kontrolom koju postavimo direktno u scenu. Da bismo to izbjegli, u scenu ćemo postaviti panel (eng. pane) koji nameće određeno raspoređivanje (eng. layout) kontrola:

```
public void start(Stage primaryStage) throws Exception {  
    StackPane pane = new StackPane();  
    pane.getChildren().add(new Button("OK"));  
    Scene scene = new Scene(pane, 200, 50);  
    primaryStage.setTitle("Button in a pane"); // Set the stage title  
    primaryStage.setScene(scene); // Place the scene in the stage  
    primaryStage.show(); // Display the stage  
}
```

Dostupni rasporedi (layouts)

- **HBox** slaže kontrole jednu pored druge horizontalno. **VBox** ih slaže vertikalno
- **FlowPane** je sličan kao HBox s tim što kontrole prelaze u sljedeći red ako ne mogu stati na scenu
- **TilePane** je sličan kao FlowPane, ali svaka kontrola zauzima isti prostor (poput pločica eng. tiles)
- **StackPane** slaže kontrole jednu preko druge, centrirano u odnosu na scenu
- **GridPane** organizuje scenu u tabelu sa fiksnim brojem redova i kolona. Zatim se mogu za svaki red/kolonu definisati dimenzije, pravila širenja itd.
- **BorderPane** organizuje scenu u pet regija: gore, dolje, lijevo, desno i središte. Unutar svake od njih možete postaviti po jednu kontrolu ili novi panel
- **AnchorPane** omogućuje da kontrole "usidrite" (eng. anchor) za jedan od rubova scene, nakon čega se zadaju X i Y koordinate u odnosu na taj rub

Deklarativni UI

- JavaFX je savremeni deklarativni framework, pa ćemo u nastavku raditi na taj način
- Default **start** metoda koju smo dobili izborom JavaFX projekta kreira scenu prema priloženom FXML fajlu:

```
public void start(Stage primaryStage) throws Exception{
    Parent root =
        FXMLLoader.load(getClass().getResource("sample.fxml"));
    primaryStage.setTitle("Hello World");
    primaryStage.setScene(new Scene(root, 300, 275));
    primaryStage.show();
}
```

- FXML je jezik za opis UI zasnovan na XMLu

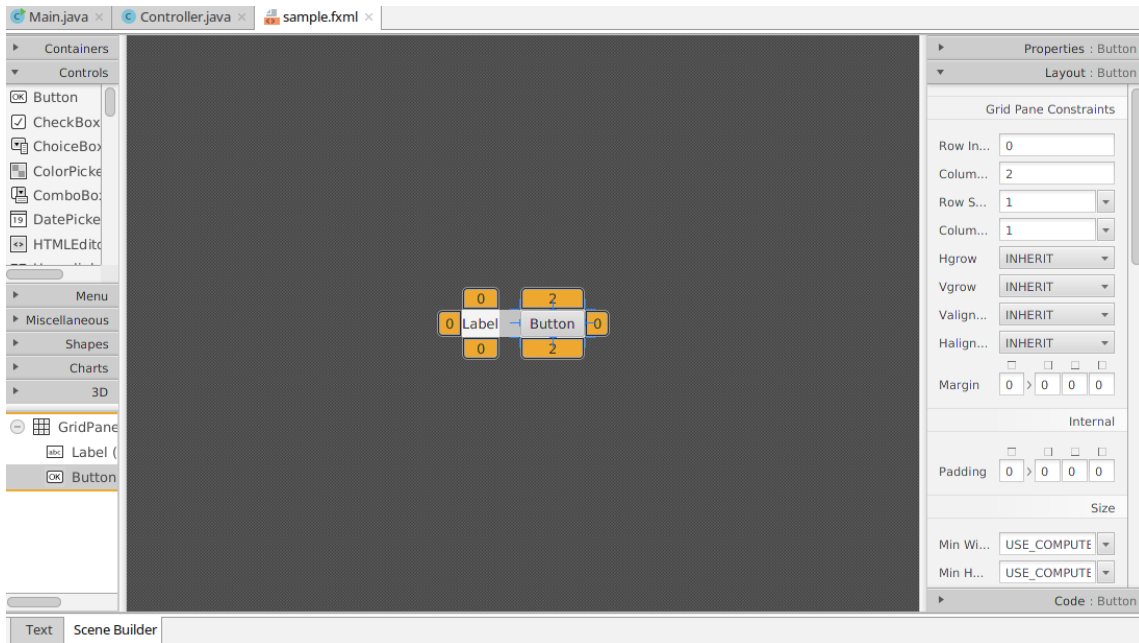
FXML

- Default generisani FXML kreira prazan GridPane u koji možemo dodavati kontrole. Controller klasa (po defaultu prazna) predviđena je da sadrži handlera za događaje

```
<?import javafx.geometry.Insets?>
<?import javafx.scene.layout.GridPane?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<GridPane fx:controller="sample.Controller"
          xmlns:fx="http://javafx.com/fxml" alignment="center"
          hgap="10" vgap="10">
</GridPane>
```

Scene builder

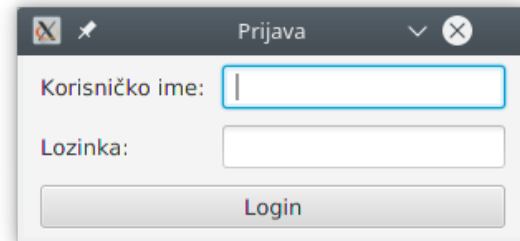
- Klikom na karticu "Scene builder" u dnu prozora (kada je selektovan fajl sample.fxml) dobijamo grafički dizajner interfejsa
- Scene builder automatski dodaje kontrole u GridPane ispod ili pored postojeće kontrole. Međutim lakše je raditi direktno sa FXMLom



Primjer

- Napraviti scenu sa izgledom kao na slici. Koriste se kontrole: **Label**, **TextField** (za lozinku **PasswordField**) i **Button**. Da bismo postigli da se dugme prostire preko dvije kolone GridPane-a koristimo property **Column Span** koji ćemo postaviti na **2**.
- Sve kontrole imaju defaultnu širinu i visinu (preferred height / width) koja ovisi od sadržaja (teksta). Da bismo postigli da dugme bude široko koliko je moguće, property **Max Width** trebamo postaviti na vrijednost **Infinity**.

Primjer: [P14_Prijava.zip](#)



- Širina scene je 300x110. Možemo spriječiti resizing prozora tako što postavimo `primaryStage.setResizable(false);`

FXML rješenje

- Kada nam dosadi igranje sa Scene Builderom možemo otkucati sljedeći FXML kod:

```
<GridPane alignment="center" hgap="10" vgap="10" xmlns="http://javafx.com/javafx/8.0.121"
xmlns:fx="http://javafx.com/fxml/1" fx:controller="sample.Controller">
  <columnConstraints> ... </columnConstraints>
  <children>
    <Label text="Korisničko ime:" />
    <TextField GridPane.columnIndex="1" />
    <Label text="Lozinka:" GridPane.rowIndex="1" />
    <TextField GridPane.columnIndex="1" GridPane.rowIndex="1" />
    <Button mnemonicParsing="false" maxWidth="Infinity" text="Login"
      GridPane.columnSpan="2" GridPane.rowIndex="2" />
  </children>
</GridPane>
```

Isto rješenje kroz Java kod

```
Label userIdLbl = new Label("Korisničko ime:");
TextField userIdTxt = new TextField();
Label userPwdLbl = new Label("Lozinka:");
PasswordField userPwdTxt = new PasswordField();
GridPane root = new GridPane();
root.setVgap(20);
root.setPadding(new Insets(10));
root.setAlignment(Pos.CENTER);

GridPane.setConstraints(userIdLbl, 0, 0);
GridPane.setConstraints(userIdTxt, 1, 0);
GridPane.setConstraints(userPwdLbl, 0, 1);
GridPane.setConstraints(userPwdTxt, 1, 1);

root.getChildren().addAll(userIdLbl,
userIdTxt, userPwdLbl, userPwdTxt);

Button signInBtn = new Button("Login");
signInBtn.setPrefWidth(Double.MAX_VALUE);
;
root.add(signInBtn, 0, 2, 2, 1);
Scene scene = new Scene(root, 250, 150);
primaryStage.setScene(scene);
primaryStage.show();
```

Deklarativno programiranje vs kod

- Da li kreirati korisnički interfejs kroz Java kod ili kroz FXML datoteku? Ovo su neke prednosti deklarativnog načina:
 - Kada se radi deklarativno, prirodnije je kreirati responsive korisnički interfejs
 - Vizuelni dizajneri interfejsa puno bolje rade sa XML datotekama nego sa kodom
 - XML se lakše edituje, manja je mogućnost greške (rjeđe se dešava da se kompletan projekat ne kompajlira), postoje dobri alati za provjeru ispravnosti XMLa
 - Kreiranje korisničkog interfejsa se može povjeriti drugom timu unutar projekta (UI tim) koji uopšte ne mora znati raditi s Java jezikom
 - Deklarativni UI nas forsira da kreiramo zasebne metode za svaku akciju u programu što olakšava kasnije testiranje projekta

Resizing GridPane

- Uočavamo da se, kada mijenjamo veličinu prozora, naš **GridPane** ne resizuje kako treba. Ovo postizemo atributima polja **ColumnConstraints** i **RowConstraints** u FXML-u:
 - Ako želimo da se širina i visina polja prilagodi širini i visini prozora, u **ColumnConstraints** dodajemo **hgrow="ALWAYS"** (slično u **RowConstraints** dodajemo **vgrow**)
 - Možemo zadati širinu i visinu polja u procentima: **percentWidth="33.3"** ili **percentHeight="20"**
 - Moguće je zadati minimalnu širinu/visinu ispod koje se dalje ne može resizovati (**minWidth...**)
 - Da bismo postigli da labele budu poravnate u desnu stranu, postavljamo u **ColumnConstraints** sljedeće: **halignment="RIGHT"**.

Stiliziranje pomoću CSSa

- Izgledom kontrola može se upravljati pomoću CSSa koji je poznat iz web aplikacija
- Desnim klikom na paket idite na **New > File** i unesite naziv **prijava.css**. Dodajte sljedeći kod:

```
#prijavaBtn {  
    -fx-background-color: lightskyblue;  
}
```
- Znak # označava da je u pitanju kontrola čiji je id "prijavaBtn". U FXML fajlu trebate Button kontroli dodati ovaj id:

```
<Button id="prijavaBtn" ... />
```

Stiliziranje pomoću CSSa

- Unutar GridPane dodajte poziv CSS datoteke:

```
<stylesheets>
```

```
  <URL value="@prijava.css" />
```

```
</stylesheets>
```

- Za polje URL trebate importovati biblioteku **java.net.URL** (a ne `javax.print.URL!!`). Sada pokrenite program i vidjećete da dugme ima plavu pozadinu.
- ID je jedinstven unutar FXML datoteke, s druge strane više kontrola može imati istu klasu (`styleClass`). Klasa se u CSSu označava tačkom umjesto znaka `#`

Stiliziranje pomoću CSSa iz koda

```
Button prijavaBtn = new Button ("Login");  
prijavaBtn.setId("prijavaBtn");
```

...

```
scene.getStylesheets().add(getClass().getResource("prijava.css  
").toExternalForm()));
```

- [Dokumentacija za JavaFX CSS](#)

Organizacija koda za resurse

- Statičke datoteke (fxml, css, slike...) možete držati u istim folderima kao i source code, ali je ovo neuobičajena i nepraktična organizacija
- Možete im pristupati i sa Interneta (preko URLova), ali onda vaša aplikacija ne bi radila ako ima problema sa konekcijom
- Uobičajeno je da se u kodu jedan folder označi kao *folder za resurse* (eng. resources). Ovi resursi će se biti zapakovani zajedno sa ostatkom projekta, ali im se pristupa preko apsolutnog puta
- Folder za resurse pravimo na sličan način kao folder za testove: desni klik na projekat **New > Directory**, unesite naziv "**resources**", desni klik na novi folder **Mark Directory As > Resources Root**

Pristup resursima

- Unutar ovog direktorija možemo kreirati poddirektorije: img, css, fxml
- Nakon što premjestimo **sample.fxml** u folder **resources/fxml**, pristupamo mu koristeći apsolutni put unutar resources foldera. Znači u Main klasi sada trebamo imati ovako:

Parent root =

```
FXMLLoader.load(getClass().getResource("/fxml/sample.fxml"));
```

Put "/fxml/sample.fxml" odnosi se na folder fxml koji se nalazi unutar foldera resources

- Slično trebamo popraviti pristup CSS datoteci iz FXMLa:

```
<stylesheets>
```

```
    <URL value="@/css/prijava.css" />
```

```
</stylesheets>
```

Upravljanje događajima

- Da bismo kreirali kod koji se treba izvršiti klikom na dugme, na **Button** kontrolu dodajemo atribut **onAction**:

```
<Button id="prijavaBtn" maxWidth="Infinity" ... onAction="#loginClick"  
text="Login" ... />
```

- Obratite pažnju na znak #! Zatim kliknite na riječ **loginClick**, pritisnite Alt+Enter i izaberite "Create method in Controller". U Controller klasi će se pojaviti metoda u koju možete upisati neki kod

```
public void loginClick(ActionEvent actionEvent) {  
    System.out.println("Dugme kliknuto!");  
}
```

Pristup poljima forme

- Da bi polja forme mogla biti dostupna iz koda, morate u FXML dodati atribut **fx:id** (ovaj atribut ujedno važi i kao **id** atribut za CSS):

```
<TextField fx:id="loginField" GridPane.columnIndex="1" />
```

- Sada kliknite na **loginField**, pritisnite Alt+Enter i izaberite "Create Field":

```
public class Controller {  
    public TextField loginField;  
    public void loginClick(ActionEvent actionEvent) {  
        System.out.println("Login glasi: " + loginField.getText());  
    }  
}
```


Promjena stila iz koda

- Želimo da se npr. u slučaju pogrešnog unosa promijeni boja pozadine polja? To možemo postići tako što definišemo posebnu CSS klasu za polje sa neispravnim unosom:

```
.poljeNijeIspravno {  
    -fx-background-color: lightpink;  
}
```

Boje **red** i **lightred** su prilično drečave, boja **lightpink** (kod mene) izgleda ok

- Analogno možete dodati i **poljeIspravno**
- Zatim u event handleru dodamo tu klasu na odgovarajuće polje:
`loginField.getStyleClass().add("poljeNijeIspravno");`

Šta se može programirati?

- Spisak svih osobina i događaja koji su dostupni za određeni tip kontrole možete dobiti na dva načina:
 - U Scene Builder-u, kada izaberete neku kontrolu s desne strane vidite osobine ispod **Properties**, a događaje možete naći u sekciji **Code**
 - Korištenjem dokumentacije npr. za klasu [Button](#) ili [TextField](#), obratite pažnju da su događaji klasifikovani pod svojstva (properties - imate metode npr. **setOnAction()** i **getOnAction()**), te da su većinom nasljeđeni pošto se u JavaFX obilato koristi nasljeđivanje kontrola

Praćenje izmjena kontrola

- Možemo li mijenjati boju pozadine tekstualnog polja dok korisnik kuca? To postićemo pomoću *Listenera*
- Da bismo kreirali listener najprije u controller klasi trebamo imati metodu **initialize** koju ručno dodajemo. Ova metoda će se izvršiti nakon kreiranja prozora, a prije ostalih metoda, što je bitno, jer će sva polja (fields) koja ste dodali u kontroler biti inicijalizirana:

@FXML

```
public void initialize() { ... }
```

Implementacija listenera

@FXML

```
public void initialize() {  
    loginField.getStyleClass().add("poljeNijeIspravno");  
    loginField.textProperty().addListener(new ChangeListener<String>() {  
        @Override  
        public void changed(ObservableValue<? extends String>  
                             observableValue, String o, String n) {  
            if (n.isEmpty()) {  
                loginField.getStyleClass().add("poljeNijeIspravno");  
            } else {  
                loginField.getStyleClass().removeAll("poljeNijeIspravno");  
            }  
        }  
    });  
}
```

Implementacija listenera - objašnjenje

- Ovaj dosta komplikovani kod nije problem, jer ga možete copy-pastovati u vaš projekat. Slijedi detaljnije objašnjenje:
 - Najprije smo u **initialize** metodi postavili stil login polja da je polje neispravno (po defaultu je neispravno, jer je prazno)
 - Deklaracija `loginField.textProperty().addListener` kreira listener (slušač) - funkciju koja će se izvršiti svaki put kada se tekst login polja promijeni
 - Zatim slijedi deklaracija tzv. anonimne klase koja u sebi sadrži funkciju koja se izvršava - dosta ružna sintaksa, kasnije ćemo vidjeti kako se ovo može znatno jednostavnije uraditi

Implementacija listenera - objašnjenje (2)

- Metoda **changed** sadrži kod koji nam treba. Metoda ima tri parametra: `ObservableValue<? extends String> observableValue, String o, String n`
Prvi ima dosta ružnu deklaraciju i neće nam trebati, a stringovi kojima smo dali imena **o** i **n** predstavljaju staru (old) i novu (new) vrijednost tekst polja
- Ako je nova vrijednost prazan string (`n.isEmpty()`) postavljamo vrijednost stila **poljeNiJeIspravno**, u suprotnom uklanjamo stil sa polja tako da polje sada ima svoj uobičajeni izgled
- Isti kod se mogao napisati još jednostavnije koristeći funkcionalni stil programiranja (sljedeći slajd)

Implementacija listenera

@FXML

```
public void initialize() {  
    loginField.getStyleClass().add("poljeNijeIspravno");  
    loginField.textProperty().addListener((observableValue, o, n) -> {  
        if (n.isEmpty()) {  
            fieldBrojStanovnika.getStyleClass().add("poljeNijeIspravno");  
        } else {  
            fieldBrojStanovnika.getStyleClass().removeAll("poljeNijeIspravno");  
        }  
    });  
}
```

Značenja
parametara kao i
ranije

Properties binding

- Da ne biste morali svaki put preuzimati kontrole sa forme, kao u primjeru ranije, možete povezati vrijednost kontrole (tekstualnog polja, labele...) sa nekim poljem u kontrolerskoj klasi. Koraci do ovoga su sljedeći:
 - U kontrolerskoj klasi dodajte privatni atribut tipa `*Property` (npr. `SimpleStringProperty`):
 - `private SimpleStringProperty login;`
 - Kreirajte sljedeće gettere:
 - ```
public SimpleStringProperty loginProperty() {
 return login;
}
public String getLogin() {
 return login.get();
}
```



## Properties binding (2)

- Novi property objekat se treba kreirati u konstruktoru. Ne zaboravite da konstruktor mora biti public!

```
public Controller() {
 login = new SimpleStringProperty("");
}
```

- U FXML datoteci postavite da odgovarajuće polje (atribut value ili text) ima vrijednost reference na property:

```
<Button id="prijavaBtn" maxWidth="Infinity" ... text="$
{controller.login}" ... />
```

---

## Properties binding (3)

- Sada u metodama možete pozivati **set** i **get** metode ovog property-ja:

```
public void loginClick(ActionEvent actionEvent) {
 System.out.println("Login glasi: " + login.get());
 login.set("default");
}
```

- Primjećujemo da, istog trena kada pozovemo metodu **set** nad atributom **login**, mijenja se vrijednost polja na formi! Na ovaj način ostvaruje se MVC organizacija koda o kojoj ćemo govoriti na jednom od narednih predavanja

## Bidirectional binding

- Ranije opisani pristup radi za read-only kontrole kao što su Label i Button, no on ne može raditi za **TextField** ili **PasswordField**
- Za povezivanje ovih kontrola potrebno je dvosmjerno uvezivanje (bidirectional binding) koje se trenutno ne može uraditi iz FXMLa, nego je potrebno izvršiti povezivanje iz Controller klase: najprije dodamo **fx:id** na **TextField**:

```
<TextField fx:id="loginField" GridPane.columnIndex="1" />
```

- Zatim u Controller klasi dodajte **initalize()** metodu u kojoj možete uspostaviti ovu vezu: (sljedeći slajd)

---

## Bidirectional binding (2)

- Zatim u Controller klasi dodajte **inititalize()** metodu u kojoj možete uspostaviti ovu vezu:

```
public TextField loginField;
private SimpleStringProperty login;
public Controller() {
 login = new SimpleStringProperty("");
}

@FXML
public void initialize() {
 loginField.textProperty().bindBidirectional(login);
}
```

---

# BorderPane

- Koji layout koristiti za glavni prozor? Do sada smo koristili GridPane i vidjeli smo da je on prilično koristan. Spomenućemo još i **BorderPane** i **AnchorPane**
  - **BorderPane** je vrlo pogodan za izradu "glavnih prozora" aplikacije. On nam omogućuava da kompletan ekran organizujemo u pet specijalnih područja: vrh, dno, lijevo, desno i centar, pri čemu se promjenom veličine prozora mijenja samo veličina centralnog dijela
  - Organizacija unutar tih dijelova se postiže drugim panelima. Npr. ako hoćemo da zona "dno" bude unutar lijeve i desne zone, u zonu "centar" postavimo drugi **BorderPane**
  - Centriranje postićemo sa **StackPane**, dok su **HBox** i **VBox** pogodni za jednostavnu organizaciju elemenata
- Primjer: [P14\\_BorderPane.zip](#)

---

# AnchorPane

- **AnchorPane** nam omogućava da postavljamo kontrole na koordinate (u pikselima, relativno od gornjeg lijevog ugla roditeljske kontrole ili nekog drugog)
  - *Nemojte nikada koristiti AnchorPane!*
- Korištenje **AnchorPane** najčešće rezultira loše dizajniranim aplikacijama, sa nabacanim, loše poravnatim kontrolama, koje ne podnose promjenu veličine (resizing). Uvijek probajte osmisliti rješenje sa kombinacijom drugih panela kako biste postigli kvalitetno dizajnirane responsive aplikacije
- *Napomena:* ako promijenite glavni panel prozora, moraćete ručno ponovo definisati atribut **fx:controller** koji određuje koja kontroler-klasa se koristi za dati FXML

# Otvaranje novog prozora

- Kod za kreiranje novog prozora (stejdža) iz JavaFX aplikacije (npr. klikom na dugme) je praktično isti kao i naša postojeća metoda **start()**
- Najprije dodajte novi FXML file desnim klikom na fxml folder i izborom **New > FXML File**.
- Tu će se crvenom bojom pojaviti naziv kontroler klase za ovaj FXML. Unesite npr. NoviController, a zatim pritisnite Alt+Enter da se ta klasa i kreira
- Alternativno, možete potpuno obrisati atribut **fx:controller** ako uopšte ne želite da imate kontroler klasu za novi stejdž  
Primjer: [P14\\_NoviProzor.zip](#)

```
<?import javafx.scene.layout.*?>
<AnchorPane xmlns="http://javafx.com/javafx"
 xmlns:fx="http://javafx.com/fxml"
 fx:controller="ba.unsa.etf.rpr.predavanje14.Novi"
 prefHeight="400.0" prefWidth="600.0">

</AnchorPane>
```

---

## Kod za otvaranje prozora

- Recimo da na prvom prozoru imamo neko dugme, da smo na njemu dodali `onAction="#otvoriNovi"` i kreirali događaj pritiskom na `Alt+Enter`. Kôd ove metode izgleda ovako:

```
public void otvoriNovi(ActionEvent actionEvent) throws Exception {
 Stage myStage = new Stage();
 Parent root =
 FXMLLoader.load(getClass().getResource("/fxml/novi.fxml"));
 myStage.setTitle("Novi prozor");
 myStage.setScene(new Scene(root, 300, 275));
 myStage.show();
}
```



---

# Kod za otvaranje prozora - objašnjenje

Objašnjenje:

- Kôd smo copy-pastali iz metode **start()** klase **Main** koja je automatski kreirana kada smo kreirali blank projekat
- Pošto je tamo **Stage primaryStage** parametar metode, ovdje smo dodali kreiranje novog stejdža

```
Stage myStage = new Stage();
```
- FXML loader baca izuzetak (konkretno ako postoje neke sintaksne greške u FXMLu ili u kontroler klasi). Zbog toga smo morali u zaglavlje metode dodati **throws Exception** (kao što je slučaj i u mainu)
- Unutar ovog koda možete staviti prepravke kakve želite npr.: dimenzije prozora, naslov prozora, da li se može promijeniti veličina itd. itd.

---

## Kod za zatvaranje prozora

- Recimo da na novom prozoru imamo dugme "Zatvori prozor" kojem je pridružena akcija `zatvoriProzorPropuhJe`. Ova akcija bi glasila:

```
public void zatvoriProzorPropuhJe(ActionEvent actionEvent) {
 Node n = (Node) actionEvent.getSource();
 Stage stage = (Stage) n.getScene().getWindow();
 stage.close();
}
```

---

## Pristup poljima novog prozora

- Recimo da je novi prozor bio formular i sada želimo da pristupimo vrijednostima formulara
- Najprije u prozoru **novi.fxml** napravimo bidirectional bind odgovarajućih polja kako je objašnjeno ranije. To znači da ćemo u kontroler klasi imati metode oblika **getNešto** koje će nam vraćati vrijednost polja (npr. string)
- Sada u kontroler klasi starog prozora moramo redizajnirati kod koji otvara novi prozor kako bismo imali pristup njegovoj kontroler klasi

---

## Pristup poljima novog prozora

```
public void otvoriNovi(ActionEvent actionEvent) throws Exception {
 Stage myStage = new Stage();
 FXMLLoader loader = new
 FXMLLoader(getClass().getResource("novi.fxml"));
 loader.load();
 noviController = loader.getController();
 myStage.setTitle("Novi prozor");
 myStage.setScene(new Scene(root, 300, 275));
 myStage.show();
}
```

---

## Pristup poljima - objašnjenje

- Koristimo drugi konstruktor klase **FXMLLoader** koji prima put to FXML fajla. Metoda **loader.load()** u tom slučaju ne vraća ništa, tako da kasnije moramo koristiti **loader.getRoot()**
- Sada imamo metodu **loader.getController()** kojom smo privatni atribut `noviController` postavili na kontroler nove forme
- Koristeći `myStage.setOnHiding` kreiramo lambda funkciju koja će se izvršiti kada se novi prozor zatvori
- Ova lambda funkcija će postaviti tekst neke labele na ono što nam je vratio `noviController.getVrijednost` (vi tu naravno možete staviti šta hoćete)

---

# Razvoj programskih rješenja

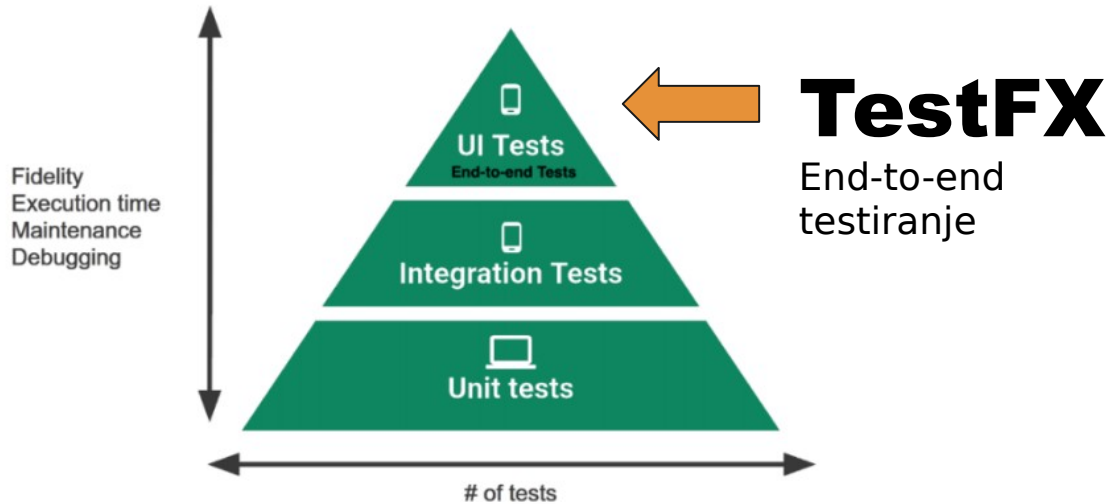
Testiranje JavaFX aplikacija koristeći  
TestFX

---

# TestFX

- Kao što smo rekli ranijim predavanjima, sve programe bi trebalo testirati. Ali kako onda testirati grafičke aplikacije? Možemo testirati pozivima metoda, pa čak i event handlera, ali koliko je takvo testiranje realno, tj. da li je to ono što će krajnji korisnik dobiti kao rezultat?
- U spas dolazi TestFX framework za pisanje aplikacija za JavaFX. TestFX radi tako što simulira klikove na kontrole, kucanje teksta i slično (možete vidjeti kursor miša kako klika po kontrolama)
- Osnovni problem sa TestFX je *manjak dokumentacije*. Verzija 4 TestFX koja je izašla prije 2-3 godine se totalno razlikuje od ranijih verzija. Osim toga, kôd TestFX testova je potpuno različit kada koristite JUnit 5 ili JUnit 4. Zbog toga je važno da kada tražite nešto na googlu navedete "testfx 4 junit 5 ..."

# Uloga TestFX





# TestFX

Da biste dodali TestFX testove u vašu aplikaciju, trebate uraditi sljedeće:

- Dodajte testnu klasu za kontrolersku klasu (Controller) prozora koji želite testirati na ranije opisan način
- Dodajte biblioteku testfx na sljedeći način: **File > Project Structure > Libraries** kliknite na + i izaberite **From Maven**. U polje za pretragu upišite **org.testfx:testfx-junit5:4.0.15-alpha** (odnosno novu verziju biblioteke)

Primjer: [P12\\_logintest.zip](#)

# Klasa ControllerTest

Dodajte manuelno sljedeće importe i deklaracije (Alt+Enter možda neće raditi):

```
import org.testfx.api.FxRobot;
import org.testfx.framework.junit5.ApplicationExtension;
import org.testfx.framework.junit5.Start;
@ExtendWith(ApplicationExtension.class)
class ControllerTest {
 @Start
 public void start (Stage stage) throws Exception {
 Parent mainNode = FXMLLoader.Load(Main.class.getResource("prijava.fxml"));
 stage.setScene(new Scene(mainNode));
 stage.show();
 stage.toFront();
 }
}
```

# Objašnjenje

- Da bi TestFX mogao testirati korisnički interfejs, moramo imati posebnu metodu u testnoj klasi (označenu anotacijom @Start) koja otvara potrebne prozore. Klasu ControllerTest proširujemo TestFX klasom kako bismo imali pristup ovim specifičnim metodama.
- U našem slučaju, samo učitavamo FXML i prikazujemo ga.
- Naredba `stage.toFront()`; osigurava da testovi neće pasti zato što je aplikacijski prozor iza IDEA prozora (npr. zato što ste upravo kliknuli na dugme za testiranje).
- Korisno je još dodati referencu na prozor (Stage) te na kontroler klasu (bilo preko `getController` kako je objašnjeno ranije ili na neki drugi način).

# Testna metoda

- Napravimo i naš prvi test:

`@Test`

```
public void testButtonClick (FxRobot robot) {
 Button btnPrijava =
 robot.lookup("#btnPrijava").queryAs(Button.class);
 robot.clickOn("#fldLogin");
 robot.write("anonymous");
 robot.clickOn("#btnPrijava");
 assertEquals("anonymous", btnPrijava.getText());
}
```

# Objašnjenje testne metode

- Testna metoda prima objekat klase FxRobot koji nam omogućuje da simuliramo različite akcije na formi (pomjeranje kursora miša, klikanje, unos teksta itd.), također nam nudi da pronalazimo kontrole preko njihovog **fx:id** atributa npr.  
`Button btnPrijava = robot.lookup("#btnPrijava").queryAs(Button.class);`
- Referenca na Button nam treba radi kasnijeg poziva metode **getText**, mada TestFX ima i "matchere" pomoću kojih se mogu očitati vrijednosti različitih labela ili kontrola, te vlastite tvrdnje (assertions)

*Pažnja: Kada koristite automatsko dovršavanje, obratite pažnju da **FxRobot** (sa malim x) nije isto što i **FXRobot** (sa velikim x) !!*

## Objašnjenje testne metode (2)

Još metoda klase FxRobot:

- **clickOn** - simulira klik mišem na kontrolu. Parametar može biti **fx:id** (naveden iza znaka povisilice #) ili tekst labele na kontroli;
- **write** - simulira kucanje teksta pomoću tastature, parametar je string koji predstavlja tekst koji treba otkucati;
- **press** - simulira pritisak jedne tipke, parametar je posebna konstanta u klasi **KeyCode** koja predstavlja id tipke;
- **release** - pritisnuta tipka se mora otpustiti (skloniti prst) što se dešava pozivom release.

## Kratice na tastaturi (shortcuti)

- Klasa FxRobot dozvoljava *lančano povezivanje* tj. svaka metoda vraća instancu klase FxRobot nad kojom možemo pozvati sljedeću metodu itd.
- Ako nam je potrebno da pritisnemo kombinaciju tipki, možemo spajati metode press i release:  
`robot.press(KeyCode.CONTROL).press(KeyCode.A).release(KeyCode.A).release(KeyCode.CONTROL);`
- Problem sa ovim kodom je to što kratice na tastaturi nisu iste na svim operativnim sistemima! Npr. na operativnom sistemu MacOSX ne postoji tipka Control nego se umjesto nje koristi tipka Command:

```
KeyCode ctrl = KeyCode.CONTROL;
if (System.getProperty("os.name").equals("Mac OS X"))
 ctrl = KeyCode.COMMAND;
robot.press(ctrl).press(KeyCode.A).release(KeyCode.A).release(ctrl);
```

# Asinhroni rad

- Obratite pažnju da je testiranje korisničkog interfejsa *asinhrono*, tj. dešava se u odvojenoj niti od glavnog programa. To znači da, za neke operacije, kada napišemo testfx kod (npr. `robot.clickOn("nesto")`) ne mora značiti da je sve što treba urađeno u sljedećoj liniji
- Ako smo npr. klikom otvorili novi prozor, moramo sačekati da kontrola postane vidljiva prije nego što koristimo `click`, `lookup` itd. u suprotnom bi test mogao pasti. Ovo čekanje postićemo na sljedeći način:

```
robot.lookup("#lblPozdrav").tryQuery().isPresent();
```
- Ako imate grešku u kodu zbog koje se prozor nikada neće otvoriti, linija iznad će jednostavno nastaviti dalje i naredni testovi će pasti. Ovo možete riješiti koristeći `assertNotNull` nad referencom koju vraća `lookup` metoda



## Asinhroni rad (2)

- Kada imate referencu na klasu, možete pozivati sve metode te klase, ali neke od tih metoda morate pozivati u pozadinskoj niti (eng. thread). To se postiže konstrukcijom

### **Platform.runLater():**

```
ComboBox comboGrad = robot.lookup("#comboGrad").queryAs(ComboBox.class);
Platform.runLater(() -> comboGrad.show());
```

- Problem je što ne znamo kada će se završiti prikazivanje menija (to može trajati reda nekoliko desetinki zbog animacije). Sljedeći kod čeka 0.2 sekundi da se meni pojavi:

```
// Čekamo da se pojavi meni sa opcijom "Sarajevo"
try {
 Thread.sleep(200); // Slobodno stavite više ako ne bude dovoljno
} catch (InterruptedException e) {
 e.printStackTrace();
}
robot.clickOn("Sarajevo");
```



## Nastavak...

- Svi zajedno učimo o TestFX platformi i otkrivamo njene nove mogućnosti
- [Zvanična dokumentacija](#) je, po mom mišljenju, prilično loša

---

# **Razvoj programskih rješenja**

Dizajn korisničkog interfejsa

---

# Značaj dizajna

- Dobro dizajniran korisnički interfejs olakšava korištenje softvera i stvara pozitivne emocije kod korisnika
- Korisnici koji su frustrirani interfejsom stvaraju negativne emocije prema proizvodu i kompaniji koja ga je isporučila



## Značaj dizajna (2)

- U računarstvu preovladava termin HCI (Human-Computer Interaction; interakcija čovjek-računar). Oblast [HCI](#) dotiče se psihologije i grafičkog dizajna, a izučava se od 80-tih godina
- Standard ISO-9421 pod nazivom "Ergonomija interakcije čovjeka i sistema" (od 1993.)
- [Upotrebljivost](#) (eng. usability) označava pogodnost softverskog proizvoda za upotrebu kroz tri kriterija: efikasnost upotrebe, jednostavnost učenja i korisničko zadovoljstvo

---

## Značaj dizajna (3)

- *Emocionalni dizajn* bavi se uticajem dizajna na emotivni doživljaj proizvoda
- Korisničko iskustvo (User Experience - UX) objedinjuje sve aspekte upotrebljivosti i emocionalnog dizajna proizvoda, sve interakcije korisnika sa kompanijom, njenim proizvodima i uslugama
- Firme koje proizvode operativne sisteme i računare se brinu za iskustvo korisnika dok koristi njihov sistem, stoga im je bitno da sve aplikacije imaju uniformisan i dobro dizajniran interfejs

# Gestalt principi

Njemački psiholozi početkom 20. vijeka postavljaju temelje ljudske percepcije.

- **Udaljenost:** Objekti koji su međusobno bliski izgledaju kao grupa.

Search Now

Stop Search

☒ P2P Search

☐ Web Search

☒ Everything

☐ Audio

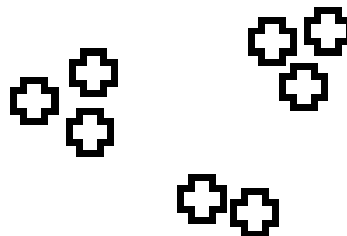
☐ Video

☐ Images

☐ Documents

☐ Software

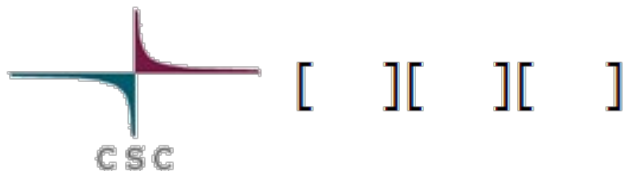
☐ Playlists



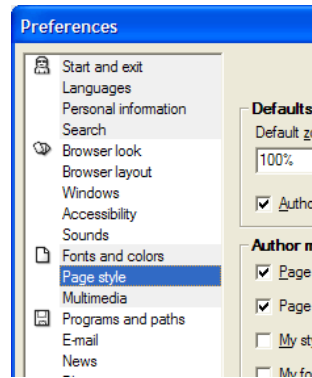
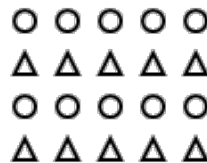
# Gestalt principi

- **Sličnost:** Objekti koji imaju slične osobine (npr. boja, veličina, oblik...) također se doživljavaju kao pripadnici iste grupe
- **Simetrija:** Ljudi nesvjesno posmatraju par simetričnih oblika kao cjelinu

Simetrija:



Sličnost:

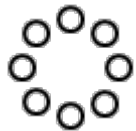




---

# Gestalt principi

- **Figura/pozadina:** Kod posmatranja kompleksnog oblika, neki objekti dolaze u prvi plan, a ostali se utapaju u pozadinu
- **Zatvarenje:** Ljudski mozak nesvjesno dovršava nepotpun objekat



---

# Pamćenje, procesiranje i učenje

- Čuvena studija iz 50-tih godina ukazuje da je čovjek u stanju da istovremeno obradi "sedam podataka - plus ili minus dva", odnosno da je to kapacitet ljudske *radne memorije*. Ovo pravilo se zove **Millerov zakon**
- Kasnije studije ukazuju da broj nevezanih cjelina s kojima čovjek može raditi odjednom je 4, ali se u mnogim situacijama podaci nesvjesno grupišu. Rad sa više od 4 predmeta zahtijeva *brojanje* što usporava rad
- Radna memorija drži podatke 15-30 sekundi nakon kojih su oni ili pohranjeni u dugoročnu memoriju ili zaboravljeni. Korisnik vrlo rijetko pohranjuje u dugoročnu memoriju informacije za koje ne smatra da su bitne

---

# Pamćenje i procesiranje: UI design

- Meniji, trake sa alatima, opcije na ekranu itd. trebaju biti grupisani u skupine od 4-7 elemenata
- Svi podaci o stanju programa trebaju biti trajno prikazani na ekranu. Npr. polje za pretragu se ne smije obrisati kada korisnik klikne na dugme za traženje
- Interaktivne upute za upotrebu sistema trebaju biti prikazane kada je moguće.
- Pravila naučena na jednom dijelu sistema treba po mogućnosti iskoristiti na drugom, tj. različiti dijelovi sistema se trebaju ponašati na sličan način
- *Konzistentnost*: također treba koristiti ono što su korisnici naučili koristeći tuđe aplikacije - kad god je moguće koristiti gotove systemske kontrole ili kontrole koje su dizajnirane da liče na systemske, a ne praviti vlastite (npr. dijaloški prozori u klasi [Alert](#))!

---

# Navigacija tastaturom

- Kod poslovnih aplikacija često je potrebno unositi ogromne količine podataka
- *Specijalisti za unos podataka* (eng. data entry specialist) su osobe koje čitavo radno vrijeme bave isključivo unosom podataka. Bitno je da se podaci mogu unijeti što brže.
- Pokazuje se da rad sa tastaturom može biti za red veličine brži od rada s mišem.
- *Svi elementi aplikacije moraju biti potpuno upotrebljivi koristeći tastaturu a ne samo miša!*
  - Kada se pritisne Enter treba se izvršiti Ok dugme (markiranje sa DefaultButton).
  - Kretanje kroz polja formulara tipkom Tab treba slijediti logiku (ići odozgo prema dolje - obratite pažnju na FXML fajl! kojim redom su kontrole u njemu navedene?)

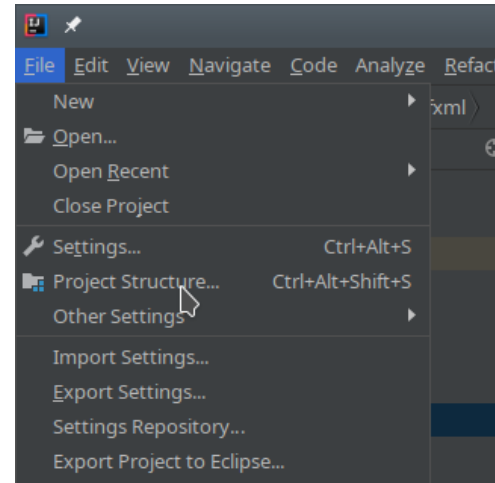
Ostatak ovog predavanja pratiće primjer: [P15\\_HCI.zip](#).

# Upotreba kontrola: Meni

- Glavni meni (eng. main menu) služi za upravljanje svim aspektima aplikacije, pored toga postoje i razni pomoćni meniji
- Glavni prozor aplikacije treba imati glavni meni, ostali prozori ne trebaju
- Značenja stavki u glavnom meniju:
  - **File:** New - novi dokument, **Open** - otvaranje datoteke, **Save** - zapisivanje datoteke, **Print** - štampa, **Exit** - kraj programa, razne opcije za uvoz/izvoz, kolaboraciju objavljivanje na webu itd. često i Settings trenutno otvorene datoteke/projekta
  - **Edit:** Undo, Redo, Cut, Copy, Paste, Find/Replace
  - **View:** opcije pogleda: prikaz raznih alatnih traka, zoom, trenutno odabrana tema
  - **Insert** i **Format:** u programima za uređivanje dokumenata (obradu teksta, crtanje...)
  - **Tools:** pozivanje eksternih alata, upravljanje dodacima (plugins)
  - **Window:** prikaz jednog ili više pomoćnih prozora
  - **Help:** pomoć za rad

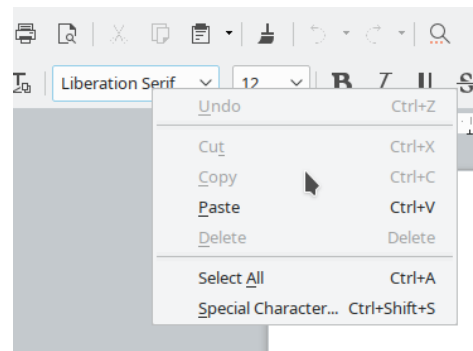
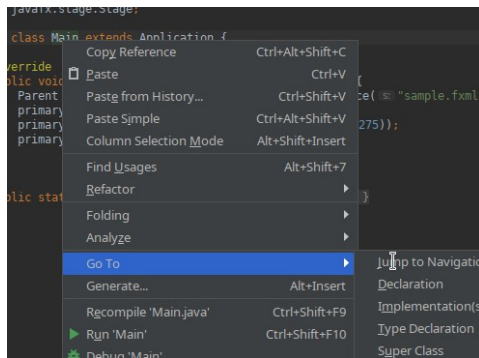
# Upotreba kontrola: Meni

- Opcije glavnog menija trebaju imati:
  - Ikonice (tamo gdje ima smisla) - npr. otvoren folder kod opcije Open
  - Meni - kratice tastature označene podvučenim slovom; npr. na slici pored vidimo da je podvučeno F u File i O u Open, što znači da ovoj opciji možemo pristupiti sa Alt+F pa Alt+O.
  - Dodatne kratice tastature: npr. za Open se obično koristi Ctrl+O, a na primjeru desno vidimo da je za Settings kratica Ctrl+Alt+S.
- **Konzistentnost!** Pogledajte kakve ikonice, kratice tastature itd. se koriste u drugim aplikacijama na vašem OS-u, posebno onim koji isporučuje ista firma kao i sam OS (Microsoft na Windowsu, Apple na OSX...)



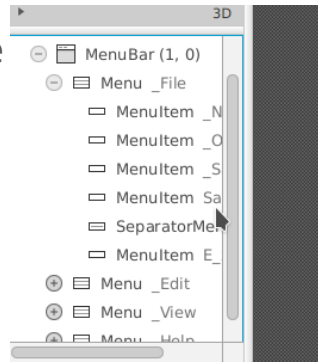
# Upotreba kontrola: Meni

- *Kontekstualni meni* (eng. context menu) dobija se klikom desnim dugmetom miša na kontrolu, nudi opcije specifične za tu kontrolu
- Kvalitetan kontekstualni meni drastično podiže upotrebljivost same aplikacije
- Meni se također može nalaziti kao padajuća lista na formi



# Pravljenje glavnog menija u JavaFX

- Da biste dodali glavni meni najprije trebate na scenu postaviti kontrolu MenuBar
- Koristimo BorderPane sekciju TOP kako bi meni bio zalijepljen za gornji dio ekrana
- Zatim u nju dodajete pojedinačne menije (kontrola Menu), a u njih stavke menija (kontrola MenuItem)
- Kontrola SeparatorMenuItem služi da u meni ubacite horizontalnu liniju
- Da biste dobili kratice tastature sa podvučenim slovom (npr. File) potrebno je da u svojstvima Menu ili MenuItem kontrole stavite "Mnemonic Parsing" na uključeno. Nakon toga u labelu jednostavno dodate underscore (donju crtu) ispred slova za koje želite da bude mnemonic npr. "E\_xit" (za Alt+X)
- Običaj je da se iza meni stavke koja rezultuje otvaranjem novog dijaloškog prozora dodaju tri tačke





# Pravljenje menija

- Dodatni shortcut (npr. Ctrl+N za opciju New) postizemo tako što na MenuItem postavljamo opciju Accelerator kroz SceneBuilder (malo je više kucanja kroz FXML).
- Akcije pridružujemo stavkama menija kao i ranije.
- Ikonu možemo postaviti: iz koda (metoda setGraphic) ili iz CSSa:

```
#mniOpen > .label{
 -fx-graphic: url("document-new.png");
}
#mniOpen .accelerator-text{
 -fx-graphic: none;
}
```

- Koristite neku od standardnih tema ikona (besplatan download). Nemojte izmišljati svoje!

# Pravljenje kontekstualnog menija

- Kreirajte instancu klase [ContextMenu](#) ili je dodajte u FXML
- ContextMenu popunite opcijama kao i inače
- Kontrolu kojoj želite da pridružite ovaj kontekstualni meni, tako da se otvara klikom na desnu tipku miša (npr. TextField) pridružite meni kroz kod pozivom metode **setContextMenu**

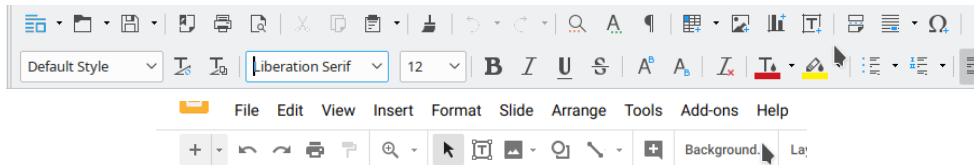
```
textField.setContextMenu(contextMenu);
```

- Da biste pridružili meni kontrolu kroz FXML morate ga ugnijezditi u **contextMenu** tag (kutija desno)

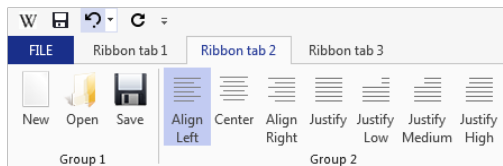
```
<TextField fx:id="tf">
 <contextMenu>
 <ContextMenu fx:id="cmTF">
 <items>
 <MenuItem text="Add"/>
 <MenuItem text="Remove"/>
 <MenuItem text="Enhance"/>
 </items>
 </ContextMenu>
 </contextMenu>
</TextField>
```

# Upotreba kontrola: Toolbar

- *Traka s alatima* (eng. toolbar) je niz ikona koje se nalaze između menija i glavnog ekrana. Tu su izdvojene neke najčešće korištene operacije



- Počevši od MS Office 2010 koristi se **ribbon** koji je nešto između menija i toolbara. UI eksperti su podijeljeni po pitanju šta je bolje. Generalno toolbars se sve rjeđe koriste



# JavaFX Toolbar

- Da bismo dodali toolbar iz TOP sekcije skidamo MenuBar, a umjesto nje stavljamo VBox u koji stavljamo MenuBar i kontejner ToolBar
  - Možete koristiti Cut/Paste da ne morate ponovo sve kreirati.
- Ikone u toolbaru su obične Button kontrole. Postavljate ikonu opet kroz CSS.
- Da razmaknete ikone koristite Separator kontrolu.
- Default Button kontrole su izdignute. Ravan izgled se može opet postići kroz CSS (vidjeti primjer - preveliko za slajd - CSS klasa .tbButton).
- Pažljivo izaberite ikone tako da je jasno šta koje dugme na toolbaru radi. Koristite standardne teme ikona, po mogućnosti neku sa flat dizajnom (ali to je stvar mode...)

# Kontrole formulara

- **Tekstualno polje** (eng. text field) služi za slobodan unos teksta.
- **Radio dugme** (eng. radio button) se koristi kada korisnik treba odabrati jednu od nekoliko opcija.
- **Dugme selekcije** (eng. check box) se koristi kada određena opcija treba biti ili uključena ili isključena.
- **Klizač** (eng. slider) omogućuje da se odabere jedna u nizu numeričkih vrijednosti npr. [1,100].
- **Padajući meni** (eng. drop down) je pomoćni meni gdje korisnik bira jednu od nekoliko vrijednosti.
- Pored njih postoje i druge specijalne kontrole npr. datumsko polje, polje za IP adresu, polje za fontove i slično. Specijalne kontrole treba koristiti što je više moguće! Koristite Google da saznate za neke inovativne specijalne kontrole.

# Kontrole formulara

**Checkbox**

Show auto-import tooltip for: ☒ Classes ☒ Static members and fields

Insert imports on paste: **Padajući meni**

- Always
- Always
- Never
- Ask

☐ Add unambiguous imports on the fly

☐ Optimize imports on the fly

Exclude from auto-import and completion:

+	-
Class, package, or member	

**Tabelarni izbor**

Use the \* wildcard to exclude all members of a specified class or package

**Radio dugme**

☐ No proxy

☐ Auto-detect proxy settings

☐ Automatic proxy configuration URL:

Clear passwords

☒ Manual proxy configuration

☒ HTTP ☐ SOCKS

**Tekstualno polje**

Host name:

Port number:  **Spinner**

No proxy for:

Example: \*.domain.com, 192.168.\*

☐ Proxy authentication

Izborom radio dugmeta, odgovarajuća grupa opcija postaje (de)aktivirana

# Kontrole formulara

Deadline (optional)

12/13/2021 00:00 +0100

December 2021

Sun	Mon	Tue	Wed	Thu	Fri	Sat
28	29	30	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	1

Datum/vrijeme kontrola  
(DateTime chooser)

Repository visibility

Izbornik boja (color  
chooser)

Drop shadow

Color

Transparency

Angle

Distance

Blur Radius

Klizač (slider)

---

## Vrste kontrola: Padajući meni

Kada kažemo "padajuća lista" to može biti jedna od nekoliko kontrola:

- **ChoiceBox** nam omogućava izbor jedne od nekoliko fiksnih opcija, dok **ComboBox** (ako aktiviramo opciju Editable) nam omogućava i da unesemo svoju vrijednost
- Također na formu možemo postaviti i kontrolu **MenuButton** koja prividno izgleda slično kao **ChoiceBox**. Međutim, **MenuButton** se puni stavkama tipa **MenuItem** koje nam nude daleko više fleksibilnosti: možemo postavljati separatore, podmenije, akceleratora i mnemonike, svaka opcija može imati svoj `onAction` itd.



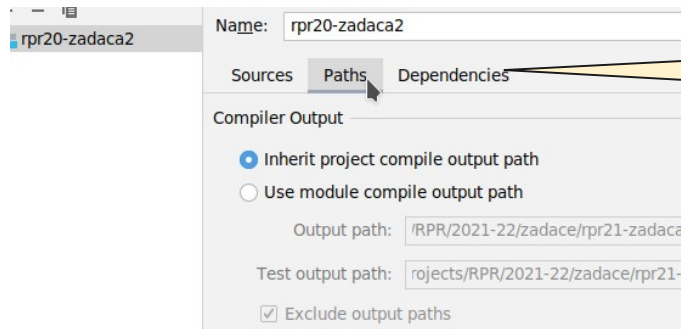
## Vrste kontrola: Liste

Kada želimo da izlistamo primjerke nekog tipa možemo koristiti:

- **ListView** (obična lista)
- **TableView** nam omogućuje da imamo više kolona na listi
- **TreeView** (pogled stabla) nam nudi pod-liste koje se otvaraju ili zatvaraju klikom na ikonicu u lijevom uglu
- Tu je i **TreeTableView** koji kombinuje obje osobine

# Kontrole za grupisanje

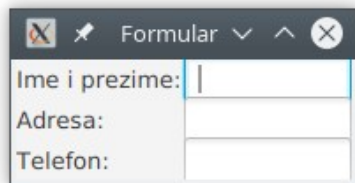
- Grupna kutija (eng. groupbox) služi za uokvirenje dijela forme. JavaFX je ne posjeduje direktno ali se može simulirati pomoću CSSa ([primjer](#))
- Kartice (eng. tabs) omogućuju da se veliki broj elemenata forme razvrstaju u nekoliko vizuelno odvojenih kategorija



Klikom na karticu Paths otvaramo grupu opcija vezanih za putanje (paths)

# Pravila dizajna formulara

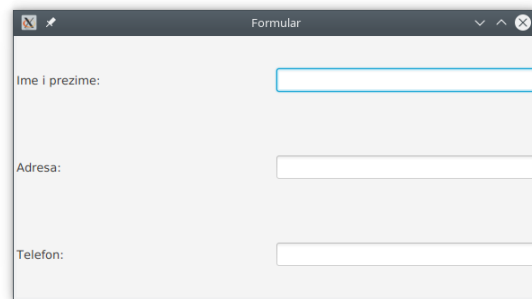
- Za dizajniranje formulara (forme) pogodno je koristiti GridPane jer će na taj način kontrole biti lijepo poravnate po redovima/kontrolama
- Osnovni problem GridPane-a je što promjena veličine prozora može rezultirati da polja formulara budu nabijena jedno do drugog (slika dolje lijevo), a kada se prozor raširi da budu previše razmaknuta (slika dolje desno). Također su kontrole priljubljene uz rub prozora što je ružno



Ime i prezime: |

Adresa:

Telefon:



Ime i prezime: |

Adresa:

Telefon:

# Pravila dizajna formulara

- Prozori sa formularima u pravilu trebaju biti fiksne dimenzije, što se postiže prilikom kreiranja Stage-a:  

```
myStage.setResizable(false);
```
- Dimenzije prozora trebaju biti minimalne koje mogu prikazati sve kontrole. Možemo koristiti konstantu `USE_COMPUTED_SIZE` kao kod pojedinačnih kontrola:  

```
myStage.setScene(new Scene(root, USE_COMPUTED_SIZE, USE_COMPUTED_SIZE));
```
- U `GridPane` kontroli obrišite **`prefWidth`** i **`prefHeight`** attribute
- Postavite **`hgap`** i **`vgap`** na 4 piksela radi većeg razmaka između kontrola
- Postavite padding sa svih strana na 10 piksela kako bi formular bio odmaknut od rubova prozora

# Još neki savjeti za formulare

- Padding FXML (lakše ga je podesiti kroz SceneBuilder):

```
<padding>
 <Insets bottom="10.0" left="10.0" right="10.0" top="10.0" />
</padding>
```

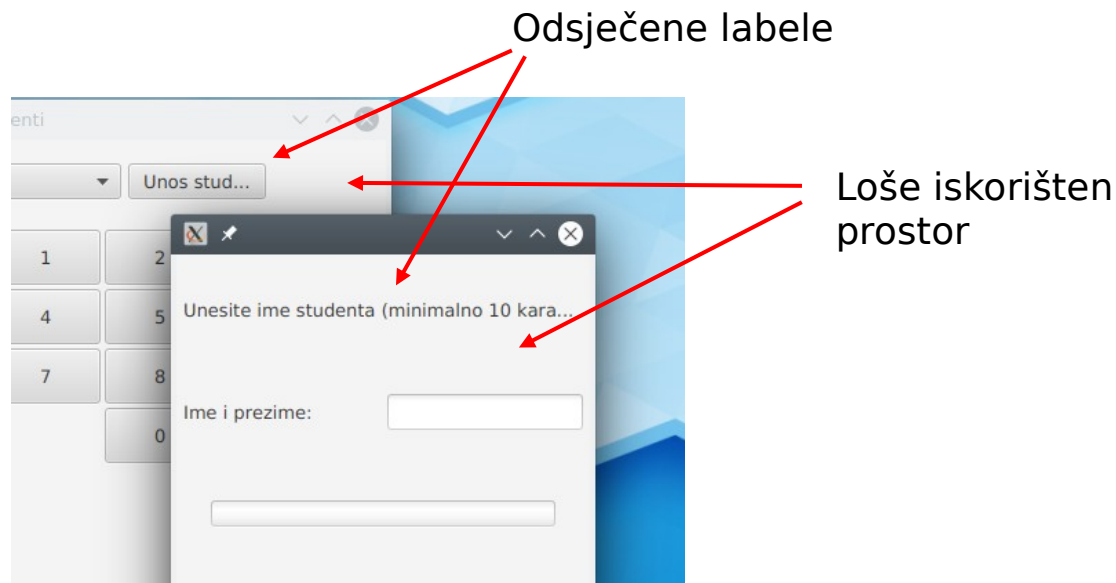
- Da biste grupisali kontrole, stavite ih u još jedan GridPane, kojem možete dodati rub kroz CSS

```
.groupBox {
 -fx-border-color: lightGrey;
 -fx-border-width: 2;
 -fx-border-radius: 2px;
}
```

# Responsiveness

- Aplikacije pravljene u 21. vijeku trebaju biti *responsive*, što znači da:
  - Treba se moći mijenjati veličina glavnog prozora aplikacije (resizable)
  - Aplikacija se treba prilagođavati promjeni veličine tako da se sav dostupni prostor iskoristi za prikaz korisnih informacija korisniku. Ne treba ubacivati prazan prostor između kontrola nego povećavati kontrole
  - Treba nastojati da nijedna informacija ne bude nedostupna korisniku
- To znači da:
  - Ne smijete koristiti AnchorPane
  - Trebate se jako dobro upoznati sa drugim vrstama panela (posebno BorderPane i GridPane)
- Ovo važi za glavni ekran - šta je sa drugim prozorima?
  - Neki od sporednih prozora trebaju biti fiksne dimenzije, posebno formulari i dijalozi

# Primjer



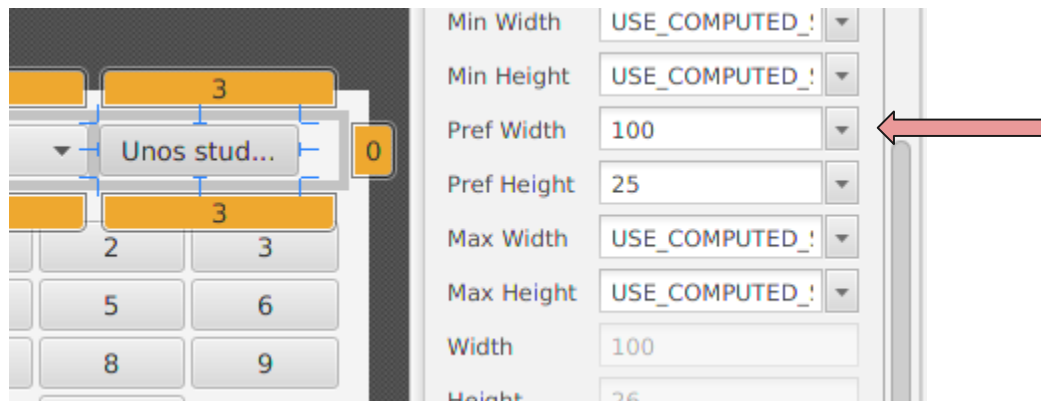
---

## Odsječene labele

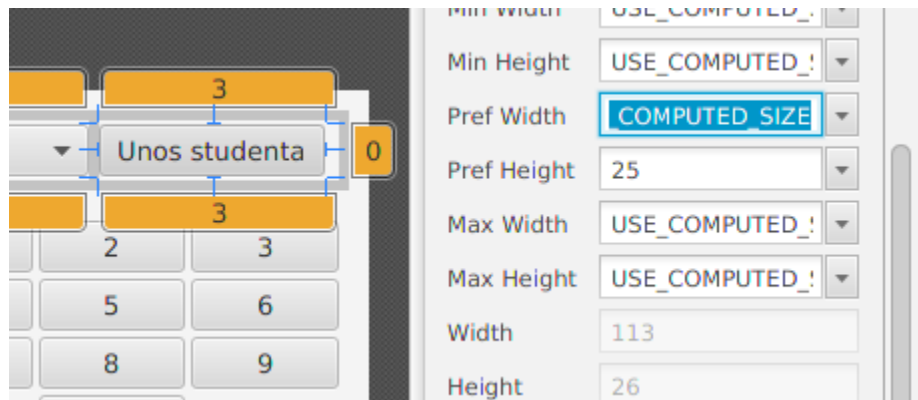
- Kada na ekranu nema dovoljno prostora da se prikaže tekst neke labele (kontrole Label ili teksta na dugmetu, meniju...), JavaFX automatski ubacuje tri tačke
- Ako se kompletan tekst na labeli ili dugmetu ne može pročitati **to je problem**, jer korisnik ne razumije šta se dešava
- To što kod vas izgleda dobro ne znači da tako izgleda kod nekog drugog!
  - Drugi korisnici mogu imati različite fontove, različitu rezoluciju, različitu temu što uzrokuje veće/manje radijuse ćoškova kontrola...
- *Kako riješiti:* Nikada nemojte postavljati dimenzije (dugmeta, prozora...) u pikselima! Dozvolite da se kontrola povećava onoliko koliko je velik prozor (USE\_COMPUTED\_SIZE). Neka korisnik sebi resizuje prozor ako ne vidi



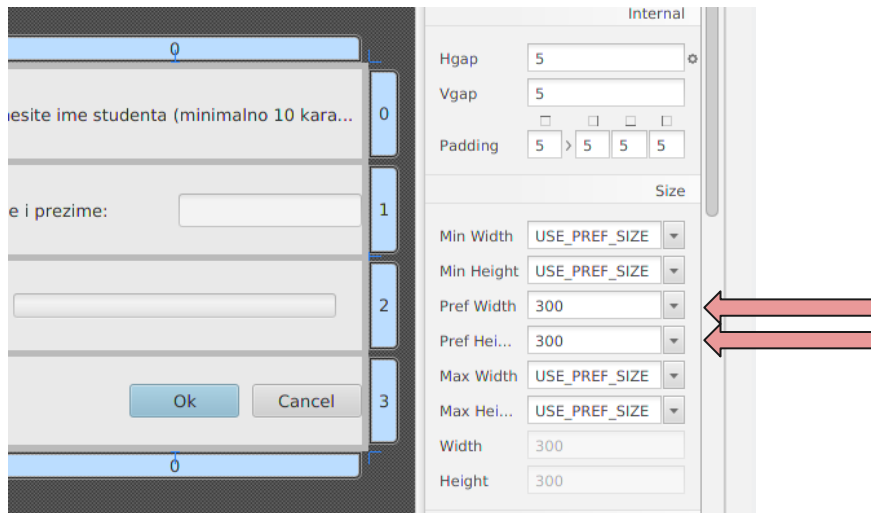
# Popravljanje odsječene labele



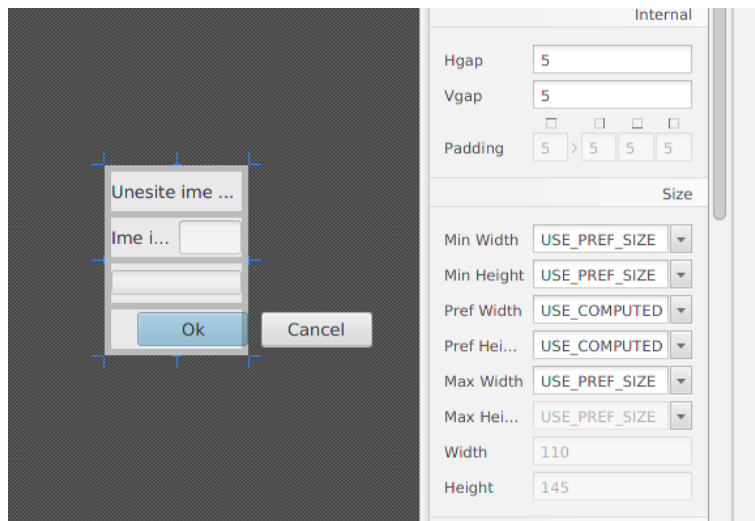
# Odsječene labele



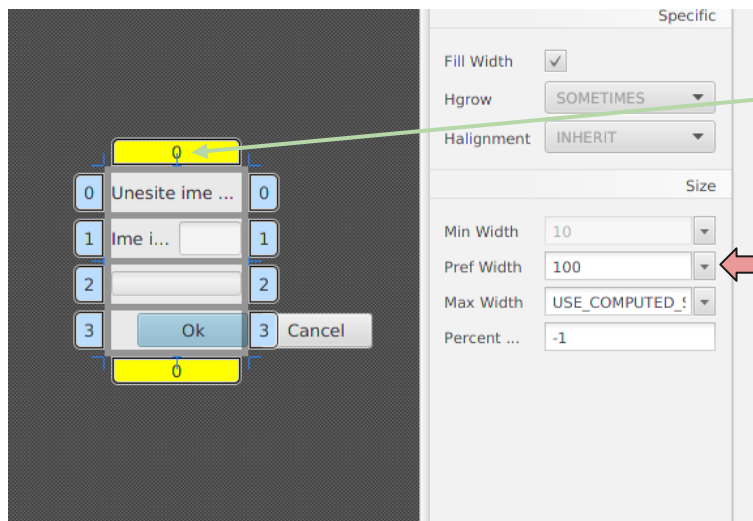
## Isto se odnosi na čitav prozor...



## Isto se odnosi na čitav prozor...

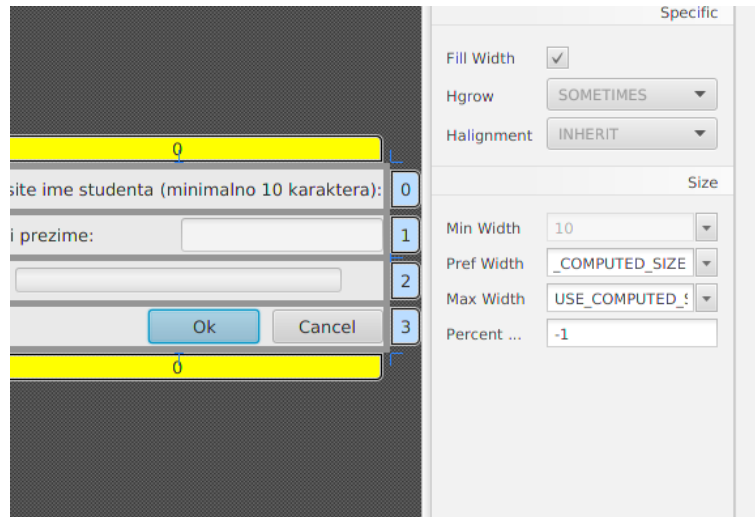


## Isto se odnosi na čitav prozor...



Klikom na zaglavlje kolone u GridPane mijenjamo organizaciju te kolone

# Isto se odnosi na čitav prozor...



---

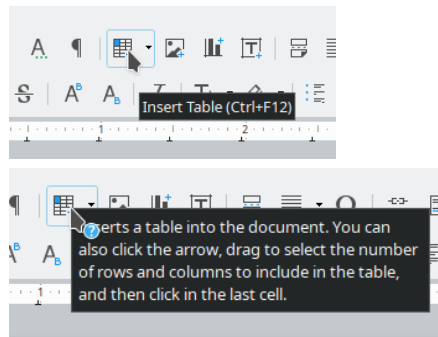
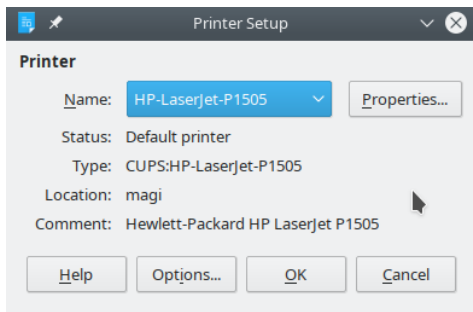
# Pomoć korisniku (Help)

Kvalitetno dizajnirana aplikacija nudi nekoliko vrsta pomoći u ovisnosti od konteksta:

- *Meni za pomoć* (Help) otvara spisak svih dokumenata o pomoći sa mogućnošću pretrage.
- *Dugme Pomoć* (Help) otvara direktno pomoć za trenutno prikazani prozor.
- *Statusna traka* (eng. status bar) je tanka traka u dnu prozora koja u svakom trenutku prikazuje šta program radi, te tako pomaže korisniku da razumije akcije, u slučaju ako se program uspori itd.
- *Savjeti* (eng. tooltips) su kratke napomene koje se dobiju kada se pokazivač miša duže zadrži na nekoj kontroli.
- *Šta je ovo?* (eng. What's This?) je posebna opcija ili ikona koja omogućuje korisniku da klikne na neki element korisničkog interfejsa i dobije detaljan opis za taj element.

# Pomoć korisniku: Primjeri

- What's This mod (označen drugačijim kursorom miša npr. upitnikom u plavom kružiću) daje više informacija od običnog tooltipa (slika desno)
- Na svakom dijaloškom prozoru nalazi se i Help dugme (slika lijevo)





## JavaFX pomoć

- Tooltip dodajete na kontrolu tako što u Sceen Builderu, u Miscellaneous sekciji prevučete kontrolu **Tooltip** na željenu kontrolu npr. TextField. Zatim u osobinama možete zadati tekst tooltipa
- Za statusnu traku, u BOTTOM sekciju **BorderPane** dodajte **HBox** u koji možete staviti labelu, a zatim CSSom postići efekat 3D "utonulog" izgleda (pogledati primjer uz predavanja, CSS klasa statusBar)

---

# Validacija polja formulara

- Jedan od najčešćih uzroka problema u informacionim sistemima su *pogrešno uneseni podaci*
- Naknadna popravka podataka je izuzetno skupa. Zbog toga treba *spriječiti korisnika* da unese neispravne podatke!
- Programer treba stalno razmišljati koji podaci uneseni u formular imaju, a koji nemaju smisla, koje su najčešće vrste grešaka sa tim konkretnim podacima i kako se eventualno mogu spriječiti
- Validacija ne može 100% garantovati da su podaci ispravni, ali kvalitetna validacija može eliminisati veliki broj grešaka

---

## Uzroci i vrste grešaka

- *Greške u tipu podataka* - npr. u polje u kojem se traži broj korisnik unosi slova. Umjesto broja 10 korisnik unese lo
- *Greške u rasponu podataka* - unesena je negativna vrijednost za težinu ili visinu, ili je unesena starost veća od 100 godina, datum u budućnosti itd.
- *Unakrsna validacija* - određena kombinacija dva ili više polja nije validna npr. ako je boja Crvena i model Mercedes, formular nije validan, jer znamo da Mercedes ne proizvodi vozila crvene boje (na primjer - ovo ustvari nije tačno)

---

# Validacija odgovarajućim tipom kontrole

Validacija polja se može *izbjeći* na način da se koriste kontrole koje ne dozvoljavaju pogrešan unos:

- Koristiti padajuće liste (ChoiceBox) kad god je moguće
- Za numeričke vrijednosti koristiti [Spinner](#) i [Slider](#) koje omogućuju da se zadaju minimalna i maksimalna vrijednost, kao i korak
- Posebne kontrole kao što je [DatePicker](#) za unos datuma (može se definisati i format datuma)
- Koristite Google da saznate da li postoje gotove kontrole ili isječki koda (snippets) za druge vrste podataka (npr. projekat [ControlsFX](#))

---

## Kada vršiti validaciju?

- *Prilikom potvrde/slanja formulara* - kada korisnik klikne na neko "Ok" dugme
- *Prilikom zatvaranja prozora*
- *Nakon unosa* - može se koristiti događaj gubitka fokusa na polju
- *Prilikom unosa* signalizirati neispravan unos

Poželjno je što ranije signalizirati pogrešan unos korisniku. Neki formulari mogu biti ogromni, dok dođe do kraja korisnik se ne sjeća šta je ranije unosio ili mora proći kroz gomilu dokumentacije da nađe taj konkretan podatak.

Najbolje je signalizirati korisniku neispravan unos odmah dok je još "svjež" i ima pri ruci sve potrebne dokumente da prekontroliše šta je i zašto unio.

---

## Kako označiti nepravilan unos?

- *Koristeći dijaloške prozore* (npr. iz klase [Alert](#)) - dijaloški prozori nisu dobra ideja jer oduzimaju fokus, dekoncentrišu korisnika i otežavaju navigaciju tastaturom.
- *Vraćanje fokusa na neispravno uneseno polje* - mada prividno bolje rješenje jer ne iskaču prozori, ponovo ponašanje je nekonzistentno i zbunjujuće. Korisnik možda neće primijetiti da se kursor vratio nazad i nastaviće kucati, te će unijeti potpuno pogrešne podatke.
- *Označavanje polja bojom* - zelena ili plava boja označavaju ispravan unos, žuta potencijalni problem (upozorenje), a crvena neispravnu vrijednost.
- *Pasivne oznake* pored polja daju detaljno objašnjenje šta je pogrešno i zašto.

Najvažnije je da korisnik ne može potvrditi i unijeti u sistem formular koji je neispravan!

## Primjer validacije prilikom unosa

- Da bismo mogli označiti bolje zelenom bojom ako je ispravno, a crvenom ako je neispravno, kreiramo CSS:

```
.poljeIspravno {
 -fx-control-inner-background: greenyellow;
}
.poljeNijeIspravno {
 -fx-control-inner-background: lightpink;
}
```

- Koristimo "control-inner-background" jer bi -fx-background uklonio i okvir oko tekstualnog polja.

## Događaj "prilikom izmjene"

- JavaFX kontrole poput `TextField` ne posjeduju događaj tipa **`onChange`**. Ovaj efekat se postiže postavljanjem slušača (listener) na `textProperty`:

```
imePrezimeField.textProperty().addListener(new ChangeListener<String>() {
 @Override
 public void changed(ObservableValue<? extends String> observableValue, String o, String n) {
 if (validnoImePrezime(n)) {
 imePrezimeField.getStyleClass().removeAll("poljeNijeIspravno");
 imePrezimeField.getStyleClass().add("poljeIspravno");
 } else {
 imePrezimeField.getStyleClass().removeAll("poljeIspravno");
 imePrezimeField.getStyleClass().add("poljeNijeIspravno");
 }
 }
});
```



# Objašnjenje

- Metoda **changed** prima staru i novu vrijednost polja (ovdje označene sa **o** za old i **n** za new)
- **validnoImePrezime** je neka funkcija koju smo napravili i možemo je pozvati da provjerimo da li je string validan
- Klasa CSS-a se ne može tek tako promijeniti. Umjesto toga imamo metode **add** i **remove** koje dodaju odnosno brišu CSS klasu sa spiska klasa primijenjenih na polje. Ako pozovemo samo **add** biće primijenjene obje klase i rezultat je nepredvidiv
- Pošto ne znamo koja funkcija je ranije pozivana i koliko puta **removeAll** će ukloniti sve primjerke klase iz spiska primijenjenih klasa

# Primjer: e-mail adresa

- Koristićemo **EmailValidator** klasu iz paketa **org.apache.commons.validator**

```
emailField.textProperty().addListener(new ChangeListener<String>() {
 @Override
 public void changed(ObservableValue<? extends String> obs, String o, String n) {
 EmailValidator validator = EmailValidator.getInstance();
 if (validator.isValid(n)) {
 emailField.getStyleClass().removeAll("poljeNijeIspravno");
 emailField.getStyleClass().add("poljeIspravno");
 } else {
 emailField.getStyleClass().removeAll("poljeIspravno");
 emailField.getStyleClass().add("poljeNijeIspravno");
 }
 }
});
```

# Validacija na promjeni fokusa

- Nekada validacija prilikom svake promjene polja može biti prespora (npr. mora se kontaktirati neki web servis da bi se provjerila validnost). U tom slučaju validacija se može vršiti kada polje *izgubi fokus* tj. kada korisnik mišem ili tastaturom pređe u neko drugo polje
- Ovo postićemo tako što jednostavno treba postaviti Listener na `focusedProperty()`. Obratite pažnju da je `ChangeListener` sada tipa `Boolean` a ne `String`:

```
imePrezimeField.focusedProperty().addListener(new ChangeListener<Boolean>() {
 @Override
 public void changed(ObservableValue<? extends Boolean> obs, Boolean o, Boolean n) {
 if (!n) ...
 }
});
```

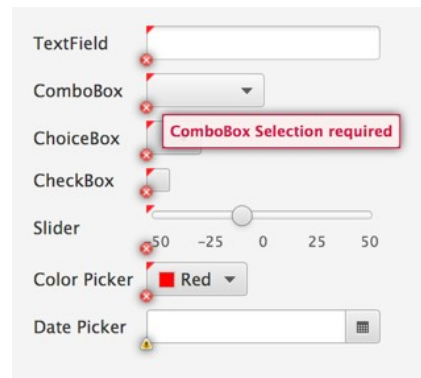
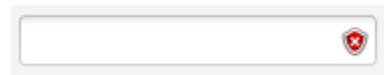
# Validacija prilikom zatvaranja prozora

- Možemo spriječiti zatvaranje prozora tako što dodamo `onCloseRequest` event na stage, što se može uraditi prilikom kreiranja stage-a. Naredba `event.consume()` sprječava zatvaranje prozora.
- Koristićemo pristup sa referencom na controller opisan u predavanju 14.

```
Stage myStage = new Stage();
FXMLLoader loader = new FXMLLoader(getClass().getResource("/fxml/formular.fxml"));
loader.load();
formularController = loader.getController();
...
myStage.setOnCloseRequest(event -> {
 if (!formularController.validan()) {
 event.consume();
 Alert alert = new Alert(Alert.AlertType.ERROR);
 alert.setTitle("Nije validno"); //...
 }
});
```

# Pasivni indikator validacije

- Pasivni indikator (na .NET platformi nazvan ErrorProvider) obično se predstavlja kao ikonica sa strane ili u ćošku kontrole u koju je unesen neispravan podatak. Klikom ili prelaskom miša preko ikonice dobija se detaljnija informacija o razlozima greške, jer je crvena boja često nedovoljna
- U sklopu [ControlsFX](#) postoji validacijski framework koji sadrži razna korisna rješenja za validaciju.



---

# **Razvoj programskih rješenja**

MVC s JavaFX

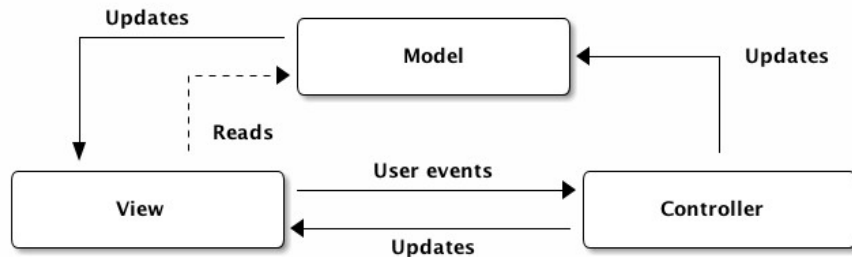
---

# Model-View-Controller

- MVC je *arhitekturni šablon dizajna* (eng. architectural design pattern) koji odgovara na pitanje organizacije koda u aplikacijama koje posjeduju UI
- Tradicionalni način rada (ubacivanje koda u event handler) rezultira loše organizovanim programom koji je težak za održavanje, krhak je (promjene UI rezultiraju nepredvidivim kvarovima funkcionalnosti), težak je za testiranje itd.
- Kod MVC sve klase su organizovane u tri skupine: Model klase, View klase i Controller klase (objekti)

# MVC model

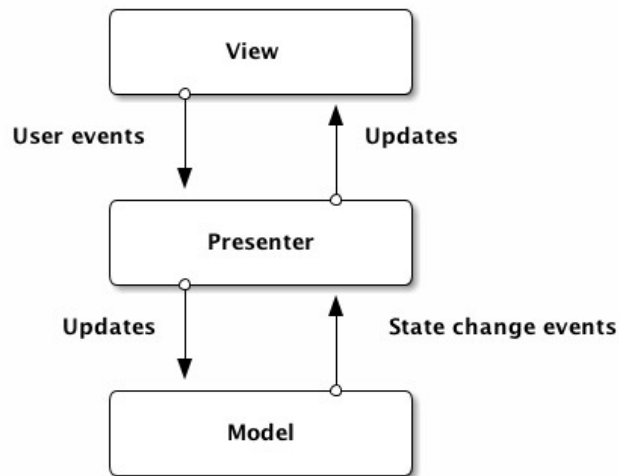
- **Model** klasa sadrži podatke i *poslovnu logiku* (eng. business logic) - to su sve formule i transformacije, uslovi validnosti itd. koji su specifični za podatke. Takođe, u modelu su sve funkcije za pristup podacima u bazi, datoteci, preko mreže itd.
- **View** predstavlja kod koji kreira korisnički interfejs (polja forme) i u principu samo prosljeđuje događaje kontroleru
- **Controller** prihvata događaje korisnika i na osnovu njih ažurira model i pogled
- Na slici puna linija označava eksplicitan poziv metode, a isprekidana implicitno ažuriranje tj. pasivnu vezu





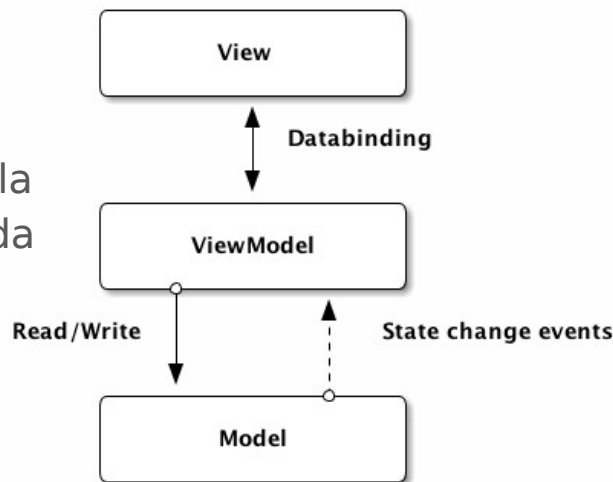
## Nedostaci MVCa i MVP

- Osnovni problem MVC-a je što je u pitanju trougao tj. svako komunicira sa svakim. Mnoge nedoumice se pojavljuju npr. na koji način mogu direktno komunicirati model i view? Da li View treba sadržavati kod za komunikaciju sa modelom?
- Zbog toga su se pojavile razne alternative MVCu
- **Model-View-Presenter (MVP)** - model ne komunicira direktno sa View-om nego Presenter proslijeđuje te informacije na View



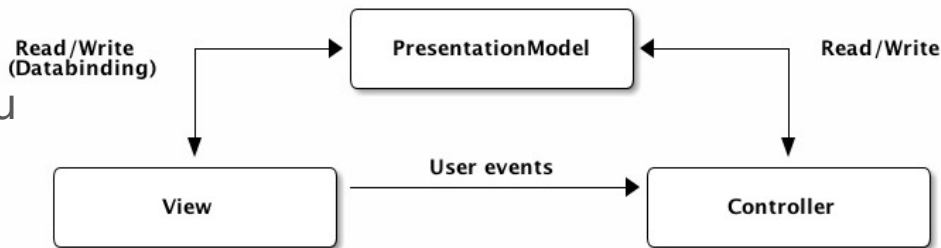
# MVVM

- **Model-View-ViewModel (MVVM)** uključuje ViewModel klase koje predstavljaju neku vrstu apstrakcije nad Modelom koja omogućava direktan databinding (pojedinačni atribut modela se može "povezati" sa UI elementom na način da ažuriranje jednog automatski ažurira drugo)
- MVVM je poznat po tome što ga koriste WPF i SilverLight frameworks koje je razvio Microsoft



# PMVC

- **PresentationModel-View-Controller (PMVC)** je unaprijeđenje MVC koje koristi najbolje osobine MVP i MVVM modela. "Prezentacijski model" je model koji podržava databinding
- Controller sada direktno može mijenjati PresentationModel kako bi podržao akcije korisnika
- Događaji korisnika se proslijeđuju kontroleru pomoću uobičajenih rukovatelja događajima (event handlera)



# Razvoj JavaFX aplikacije po PMVC modelu

- U nastavku ćemo razviti jednostavnu aplikaciju koja implementira neku vrstu MVC (ili PMVC) arhitekture, kao primjer preporučenog pristupa
- JavaFX framework kao takav je fleksibilan i dozvoljava različite arhitekture, nijedna nije preferirana
- Postoji nekoliko UI frameworks razvijenih za JavaFX koji uvode striktniju i preciznije definisanu organizaciju koda po nekom od opisanih modela (kaže se da su "opinionated", imaju svoj preferirani način rada). To su: [Griffon](#), [JRebirth](#), [afterburner.fx](#) i druge

## Povezivanje (binding)

- Do sada smo radili na sljedeći način: kada dodamo polja na formu koristili smo akcije kontrolera (npr. `onAction` nekog dugmeta) da preuzmemo vrijednosti iz polja forme i pohranimo ih u neke promjenljive. Prilikom kreiranja forme možemo iz promjenljivih napuniti vrijednosti u formu. Kod kompleksnih formi to proizvodi veliku količinu šablonskog koda u kojem se često dešavaju greške.
- Da ne biste to morali raditi, možete povezati vrijednost kontrole (tekstualnog polja, labele...) sa nekim poljem u kontrolerskoj klasi. Na taj način kada se promjenljiva promijeni automatski se mijenja polje forme.
- Ovo se naziva *povezivanje* (eng. *binding*). Postoje dva tipa povezivanja koja ćemo sada naučiti

# Jednosmjerno povezivanje

*Jednosmjerno povezivanje* (eng. unidirectional binding) nam omogućava da kada promijenimo vrijednost promjenljive u kodu, automatski se ažurira forma

- U kontrolerskoj klasi dodajte privatni atribut tipa `*Property` (npr. `SimpleStringProperty`):

```
private SimpleStringProperty username;
```

- Kreirajte sljedeće gettere:

```
public SimpleStringProperty usernameProperty() {
 return username;
}

public String getUsername() {
 return username.get();
}
```

## Jednosmjerno povezivanje (2)

- Novi property objekat se treba kreirati u konstruktoru kontrolerske klase. Ne zaboravite da konstruktor mora biti public!

```
public Controller() {
 username = new SimpleStringProperty("");
}
```

- U FXML datoteci postavite da željeni atribut (value ili text) odgovarajućeg polja ima vrijednost reference na property:

```
<Button id="prijavaBtn" maxWidth="Infinity" ... text="$
{controller.username}" ... />
```

## Jednosmjerno povezivanje (3)

- Sada u metodama možete pozivati **set** i **get** metode ovog property-ja:

```
public void loginClick(ActionEvent actionEvent) {
 System.out.println("Login glasi: " + username.get());
 username.set("default");
}
```

- Primjećujemo da, istog trena kada pozovemo metodu **set** nad atributom **username**, mijenja se vrijednost polja na formi!



## Dvosmjerno povezivanje

- Ranije opisani pristup radi za read-only kontrole kao što su **Label** i **Button**, no on ne može raditi za **TextField** ili **PasswordField**. Za povezivanje ovih kontrola potrebno je dvosmjerno uvezivanje (bidirectional binding) koje se trenutno ne može uraditi iz FXMLa, nego je potrebno izvršiti povezivanje iz Controller klase
- Najprije dodamo **fx:id** na **TextField**:
- `<TextField fx:id="userNameField" GridPane.columnIndex="1" />`
- Zatim u Controller klasi dodajte **inititalize()** metodu u kojoj možete uspostaviti ovu vezu: (sljedeći slajd)

## Dvosmjerno povezivanje (2)

- Zatim u **Controller** klasi dodajte **initialize()** metodu u kojoj možete uspostaviti ovu vezu:

```
public TextField userNameField;
private SimpleStringProperty userName;
public Controller() {
 userName = new SimpleStringProperty("");
}
```

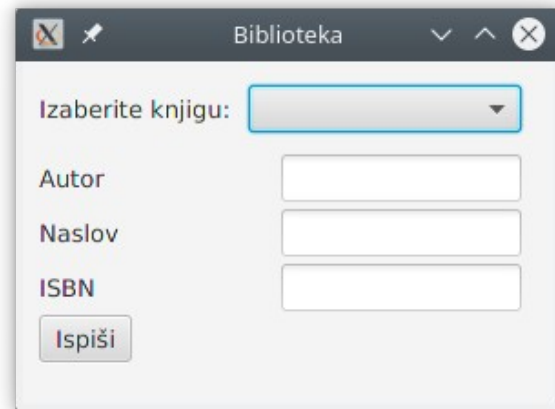
@FXML

```
public void initialize() {
 userNameField.textProperty().bindBidirectional(userName);
}
```

# Postavka problema

- Cilj je razviti aplikaciju "Biblioteka" koja omogućava korisniku da izabere knjigu koristeći **ChoiceBox** kontrolu
- Nakon izbora knjige popunjavaju se polja za Autora, Naslov i ISBN knjige, koja omogućavaju i editovanje ovih vrijednosti
- Klikom na dugme Ispiši u konzoli se ispisuje spisak svih knjiga, čime se uvjeravamo da je knjiga uspješno editovana

Primjer: [P16\\_Biblioteka.zip](#)



# JavaBeans

- Prvi korak u MVC projektu je kreirati JavaBeans
- JavaBeans (zrna kafe) su jednostavne podatkovne klase koje nemaju ništa osim atributa i settera/gettera. Po [JavaBeans specifikaciji](#) (koju smo do sada pratili), setteri se moraju zvati setAtribut, getteri getAtribut (osim ako je tip boolean u kojem slučaju isAtribut), klasa mora imati konstruktor bez parametara i implementirati interfejs Serializable
- POJOs (Plain Old Java Objects) opisuju podatkovne klase koje ne moraju nužno slijediti JavaBeans specifikaciju, ali imaju otprilike sličnu filozofiju

# JavaFX Beans

- Da bismo izveli PMVC kao napredniju verziju MVC, potrebno je da uspostavimo direktnu vezu (databinding) atributa Modela s View-om. Da bi ovo bilo moguće, moramo koristiti specijalne tipove koji se zovu *\*Property*, a nalaze se u paketu sa prikladnim imenom `javafx.beans`.
- Npr. za stringove koristimo **`SimpleStringProperty`**, a za cijele brojeve **`SimpleIntegerProperty`**
- Za početak kreiraćemo bean Knjiga:

```
public class Knjiga {
 private SimpleStringProperty autor;
 private SimpleStringProperty naslov;
 private SimpleStringProperty isbn;
 private SimpleIntegerProperty brojStranica;
}
```

# Kreiranje settera/gettera

- Za svaki property trebamo kreirati tri metode: Dvije su uobičajeni getter i setter koje mijenjaju property pozivajući njegove metode **get** i **set**, a treća metoda vraća direktno referencu na property. Ove metode su nam potrebne za dvosmjerno povezivanje. Npr:

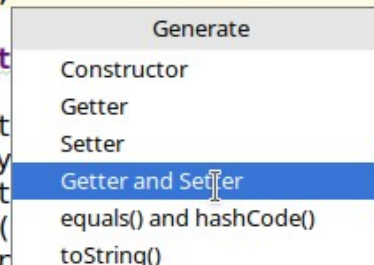
```
public String getNaslov() {
 return naslov.get();
}
public void setNaslov(String naslov) {
 this.naslov.set(naslov);
}
public SimpleStringProperty naslovProperty() {
 return naslov;
}
```

# Kreiranje settera/gettera

- IntelliJ IDEA generator setter/getter metoda slijedi JavaFX Beans specifikaciju, tako da ih možemo kreirati automatski. Najlakši način je da pritisnemo **Alt+Insert** kako bismo dobili "Generator" kontekstni meni i izabrali Getter/Setter. Sve metode bi trebalo označiti sa **final**

```
private SimpleStringProperty autor;
private SimpleStringProperty naslov;
private SimpleStringProperty isbn;
private SimpleIntegerProperty brojStranica;
```

```
public Knjiga(String a, String n, String i, int b) {
 autor = new SimpleStringProperty(a);
 naslov = new SimpleStringProperty(n);
 isbn = new SimpleStringProperty(i);
 brojStranica = new SimpleIntegerProperty(b);
}
```



# Konstruktor

- Pored defaultnog konstruktora (koji ne mora da radi ništa) dodaćemo i konstruktor koji setuje sve attribute radi bržeg kreiranja objekata:

```
private SimpleStringProperty autor = new SimpleStringProperty("");
private SimpleStringProperty naslov = new SimpleStringProperty("");
private SimpleStringProperty isbn = new SimpleStringProperty("");
private SimpleIntegerProperty brojStranica = new SimpleIntegerProperty(0);
public Knjiga() {}
public Knjiga(String a, String n, String i, int b) {
 autor = new SimpleStringProperty(a);
 naslov = new SimpleStringProperty(n);
 isbn = new SimpleStringProperty(i);
 brojStranica = new SimpleIntegerProperty(b);
}
```



---

## Model klasa

- Na osnovu ovih beansa možemo napraviti model klasu **BibliotekaModel** koja sadrži kolekciju Knjiga (npr. ova kolekcija se može dobijati iz baze podataka ili iz datoteke)
- JavaFX kolekcija koja podržava databinding je **ObservableList**. Pomoću factory metode možete kreirati razne vrste kolekcija npr. **ObservableArrayList**
- Uvešćemo i atribut koji nam označava trenutno izabranu knjigu na listi, koji je tipa **ObjectProperty**. Ova klasa nam omogućava da napravimo generički Property omotač oko bilo koje druge klase
- Na kraju trebamo napraviti i settere/gettere za trenutnu knjigu, te getter za **ObservableList** (sljedeći slajd)

---

# Model klasa

```
public class BibliotekaModel {
 private ObservableList<Knjiga> knjige = FXCollections.observableArrayList();
 private ObjectProperty<Knjiga> trenutnaKnjiga = new SimpleObjectProperty<>();
 public ObjectProperty<Knjiga> trenutnaKnjigaProperty() {
 return trenutnaKnjiga;
 }
 public Knjiga getTrenutnaKnjiga() {
 return trenutnaKnjiga.get();
 }
 public void setTrenutnaKnjiga(Knjiga k) {
 trenutnaKnjiga.set(k);
 }
 public ObservableList<Knjiga> getKnjige() {
 return knjige;
 }
}
```

# Controller klasa

- Sada možemo preći na pravljenje **Controller** klase. Prema (P)MVC modelu Controller klasa mora imati referencu na model klasu:

```
public class BibliotekaController {
 private BibliotekaModel model;
```

- Ali kako će ona dobiti tu referencu? Uobičajeni pristup je *dependency injection* što znači da konstruktor kontroler klase prima referencu na model:

```
 public BibliotekaController(BibliotekaModel m) {
 model = m;
 }
```

no problem je što onda ne možemo automatski pridružiti kontroler kroz FXML nego moramo promijeniti main funkciju. Osim toga, automatsko generisanje metoda sa Alt+Enter nam neće više raditi. Zato ćemo to ostaviti za kraj

# Povezivanje ChoiceBox sa ObservableList

- Povezivanje kontrola s podacima (databeling) se vrši u specijalnoj metodi **initialize** koju ćemo dodati u kontroler. Ova metoda se poziva nakon kreiranja interfejsa
- ChoiceBox (padajuća lista) se može povezati sa **ObservableList** preko metode **setItems**. No obratite pažnju da, nakon što dodamo **fx:id** na **ChoiceBox** i kreiramo polje, trebamo označiti **ChoiceBox** kao generičku klasu tipa Knjiga, jer je to i tip **ObservableList**-e!

```
public class BibliotekaController {
 private BibliotekaModel model;
 public ChoiceBox<Knjiga> izborKnjige;
 @FXML
 public void initialize() {
 izborKnjige.setItems(model.getKnjige());
 }
}
```

# Mocking

- Ako sada pokrenemo program, padajuća lista će biti prazna, jer nemamo nijednu knjigu u biblioteci. Za potrebe testiranja, napunićemo našu listu nekim izmišljenim podacima. U klasu **BibliotekaModel** dodat ćemo metodu **napuni()** koja obavlja taj posao:

```
void napuni() {
 knjige.add(new Knjiga("Meša Selimović", "Tvrđava", "abcd", 500));
 knjige.add(new Knjiga("Ivo Andrić", "Travnička hronika", "abcd", 500));
 knjige.add(new Knjiga("J. K. Rowling", "Harry Potter", "abcd", 500));
}
```

# Pokretanje programa

- Da bismo mogli isprobati kako sada radi uvezivanje liste knjiga, moramo dovršiti prepravku **start()** metode tako da se prije poziva **load()** funkcije klase FXMLLoader poveže najprije **Controller** s **Modelom**, a zatim **View** s **Controllerom**
- Prethodno u FXML fajlu moramo obrisati atribut **fx:controller="neki.paket.BibliotekaController"**. Npr. ako nam se na najvišem nivou nalazi neki GridPane, možda smo imali ovakav kod:

```
<GridPane alignment="center" hgap="10" vgap="10"
xmlns="http://javafx.com/javafx/8.0.121" xmlns:fx="http://javafx.com/fxml/1"
fx:controller="sample.BibliotekaController" >
```

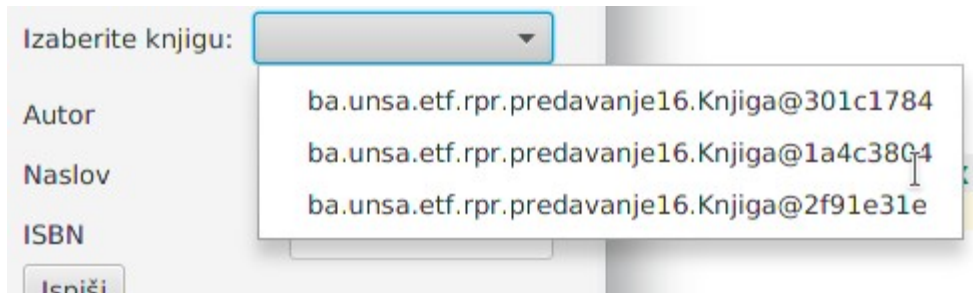
# Pokretanje programa

- Sada main treba da glasi ovako:

```
@Override
public void start(Stage primaryStage) throws Exception{
 BibliotekaModel model = new BibliotekaModel();
 model.napuni();
 FXMLLoader loader = new FXMLLoader(getClass().getResource("biblioteka.fxml"));
 loader.setController(new BibliotekaController(model));
 Parent root = loader.load();
 primaryStage.setTitle("Biblioteka");
 primaryStage.setScene(new Scene(root, 300, 275));
 primaryStage.show();
}
```

# toString metoda

- Kada pokrenemo program (ako smo do sada sve radili ispravno) ugledaćemo ovo:



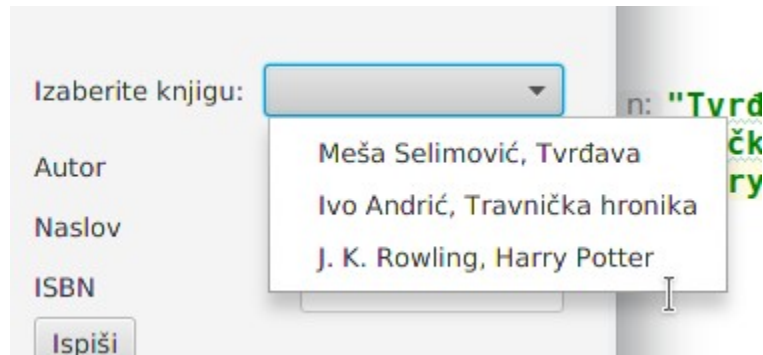


## toString metoda

- Ako hoćemo npr. da se u listi ispiše "Ime autora, Naslov knjige", možemo u klasi **Knjiga** dodati metodu **toString** koja radi upravo to:

`@Override`

```
public String toString() {
 return autor.get() + ", " +
 naslov.get();
}
```



# Trenutna knjiga

- Da bismo postigli da se polja forme ažuriraju na osnovu izabrane knjige, moramo uraditi dvije stvari u kontroleru:
  - Promjena knjige u ChoiceBox treba da mijenja atribut trenutnaKnjiga u modelu
  - Polja TextField treba dvosmjerno uvezati (bidirectional bind) sa odgovarajućim atributima trenutne knjige
- Prvi dio zadatka je jednostavan, dodamo onAction atribut na ChoiceBox i implementiramo odgovarajuću metodu na kontroleru:

```
public void promjenaKnjige(ActionEvent actionEvent) {
 model.setTrenutnaKnjiga(izborKnjige.getValue());
}
```

## Povezivanje kontrola

- Po analogiji mogli bismo pomisliti da se poziva **bindBidirectional** na trenutnu knjigu (pozivom **getTrenutnaKnjiga**). Inače bind se treba nalaziti također u metodi

**initialize**

```
public TextField knjigaAutor;
```

```
// ...
```

```
@FXML
```

```
public void initialize() {
```

```
 izborKnjige.setItems(model.getKnjige());
```

```
 knjigaAutor.textProperty().bindBidirectional(model.getTrenutnaKnjiga().autorProperty());
```

- No ovo neće raditi tj. polja forme se neće promijeniti kada se promijeni izabrana knjiga, jer sjetimo se, atribut **trenutnaKnjiga** je samo referenca na jednu od knjiga u kolekciji

## Slušači (listeners)

- JavaFX properties imaju osobinu da se na njih mogu "zakačiti" **listeners** - lambda funkcije koje će biti pozvane u trenutku kada se taj property promijeni
- Ranije smo pridruživali listenere na polja forme, ali ako obratite pažnju ustvari smo ih povezivali sa property-jem koji sadrži vrijednost polja forme
- Ispravan pristup je da na property **trenutnaKnjiga** zakačimo listener koji će povezati **TextField** kontrole sa atributima. U slučaju da su kontrole ranije bile povezane, potrebno je pozvati najprije metodu **unbind**. Srećom, lambda funkcija prima referencu na staru i novu vrijednost property-ja
- Dodavanje listenera ćemo, ponovo, postaviti u metodu **initialize()**
- Na slijedećem slajdu možete vidjeti primjer koda za polje "autor", a možete sličan kod dodati i za ostala polja. Naravno, sve se može staviti u istom listeneru

## Slušači (listeners)

```
model.trenutnaKnjigaProperty().addListener((obs, oldKnjiga, newKnjiga) -> {
 if (oldKnjiga != null) {
 knjigaAutor.textProperty().unbindBidirectional(
 oldKnjiga.autorProperty());
 }
 if (newKnjiga == null) {
 knjigaAutor.setText("");
 }
 else {
 knjigaAutor.textProperty().bindBidirectional(
 newKnjiga.autorProperty());
 }
});
```

# Organizacija složenijeg projekta u pakete

- Složenije MVC aplikacije mogu imati mnogo klasa, koje bi bilo poželjno organizirati u pakete
- Jedna preporuka za organizaciju projekta:
  - **ba.unsa.etf.rpr.projekat.model** - modelske klase
  - **ba.unsa.etf.rpr.projekat.controller** - kontroleri
  - **ba.unsa.etf.rpr.projekat.beans** - bean klase
  - **ba.unsa.etf.rpr.projekat.dal** - data access layer (pristup bazi, datotekama i sl.)
- Pošto se **View** nalazi u fxml i css datotekama koje se nalaze ispod **resources** foldera, neka posebna organizacija nije potrebna
- U slučaju još složenijih aplikacija mogu se kreirati paketi koji se tiču različitih dijelova aplikacije, od kojih svaki ima svoje **Model**, **Controller** itd. pod-pakete

---

# **Razvoj programskih rješenja**

Rad s bazom podataka

Pored konektora koji su u formi eksterne biblioteke, postoje i konekcije koje se mogu instalirati na klijent ili server (ili obje lokacije) ili konekcija preko ODBC (sistemskog) drivera

---

## Rad s bazama iz Java jezika

- Podsystem za rad sa bazama podataka naziva se JDBC i sastavni je dio Java SDK
- Međutim, driveri za rad sa pojedinačnim bazama nisu instalirani i trebaju se posebno downloadovati. Ovi driveri se nazivaju *konektori*
- Pored konektora koji su u formi eksterne biblioteke, postojali su u prošlosti konektori koji su se morali instalirati na klijent ili server (ili obje lokacije) ili konekcija preko ODBC (sistemskog) drivera
- Primjer: [P20\\_PristupBazi.zip](#) - sadrži pristup SQLite bazi uključenoj u projekat te zakomentaran kod za pristup MySQL bazi.



---

# Kako pristupiti bazi podataka?

Da biste izvršili upit na bazu podataka iz Java programa, potrebni su minimalno sljedeći koraci:

- Download i uključivanje JDBC drivera u projekat
- **Class.forName** (nije uvijek obavezno)
- Određivanje URL-a konekcije
- Kreiranje objekta Connection:  
`Connection conn = DriverManager.getConnection(url);`
- Kreiranje objekta Statement:  
`Statement stmt = conn.createStatement();`
- Poziv funkcije **executeQuery** koja prima SQL upit, a vratiti će objekat tipa ResultSet:  
`ResultSet result = stmt.executeQuery(upit);`

---

# Pristup MySQL serveru

- Da biste instalirali MySQL server na Windows računar, idite na link [www.mysql.com](http://www.mysql.com) izaberite Downloads > Community Edition > MySQL Community Server
- Na Linuxu možete instalirati kroz upravljanje paketima distribucije (npr. **apt install mysql-server**)
- U laboratoriji je instaliran MySQL na računarima. Osim toga, možete se konektovati na Oracle server koji koristite za predmet Osnove baza podataka
- Downloadujte i instalirajte [MySQL Workbench](#) grafički alat za upravljanje MySQL bazom, kreiranje tabela, administraciju itd.

---

# Prvi MySQL projekat

- Da biste dodali driver, idite na **File > Project Structure**, kliknite na + pa **From Maven...** te u polje za koordinate biblioteke upišite: `mysql:mysql-connector-java:8.0.22`
- Potrebno je navesti: `Class.forName("com.mysql.cj.jdbc.Driver");`
- URL koji trebate navesti **DriverManageru** je oblika: `jdbc:mysql://server/šema`
  - U slučaju greške **The server time zone not recognized**, dodati `?serverTimezone=UTC`
- Pod **server** trebate navesti domensko ime ili IP adresu servera na koji se želite konektovati
- Pod **šema** navodite šemu (bazu) u kojoj se nalaze tabele sa kojima želite raditi, mada to nije obavezno, nego možete navesti šemu unutar svakog upita prije naziva tabele
- Ako je potrebno korisničko ime, koristite alternativnu varijantu funkcije **getConnection**:  
`Connection conn = DriverManager.getConnection(url, "username", "password");`

---

# Rad sa ResultSet objektom

- Statement se kreira na uobičajen način:

```
Statement stmt = conn.createStatement();
```

- **ResultSet** je kolekcija koja sadrži rezultate upita, ali zbog načina kako JDBC funkcioniše efikasnije je da prolazimo kroz rezultate red po red nego da ih preuzmemo sve odjednom
- Metoda **next()** vraća sljedeći red rezultata, ili **false** ako nema više rezultata
- Metode **getInt**, **getString**... vraćaju jedan rezultat koji se konvertuje u odabrani tip. Parametar je redni broj kolone sa rezultatom (počevši od 1) ili naziv kolone

```
ResultSet result = stmt.executeQuery(upit);
```

```
while(result.next()) {
```

```
 String naziv = result.getString(1);
```

```
 float cijena = result.getFloat(2);
```

```
 ...
```

---

# SQLite baza podataka

- SQLite je "ugrađena baza podataka", što znači da je možete isporučiti uz vaš program, jer je kompletna sadržana u jednom fajlu. Intenzivno se koristi na Android OSu.
- Postoje i "memorijske baze" koje se nalaze u memoriji tj. uopšte nisu trajno smještene
- Za rad sa SQLite trebate uključiti biblioteku (From Maven): **org.xerial:sqlite-jdbc:3.34.0**
- `Class.forName` nije potreban, ali možete ga navesti:  
`Class.forName("org.sqlite.JDBC");`
- URL je oblika: **jdbc:sqlite:filename** gdje je **filename** kompletan put do datoteke. Put je relativan na korijen vašeg projekta, a možete koristiti i apsolutni put.

---

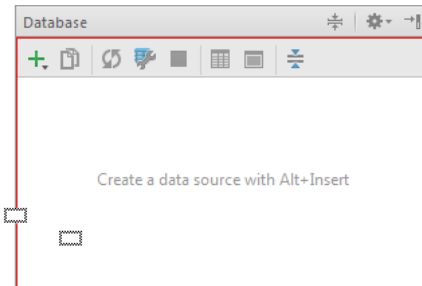
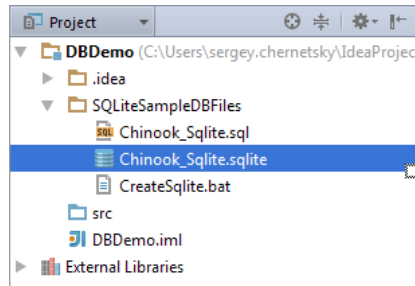
# Gdje držati SQLite bazu?

- Ako URL glasi `"jdbc:sqlite:baza.db"` fajl `"baza.db"` se nalazi u početnom direktoriju projekta (u vašem okruženju to je na istom nivou kao `src`, `out` itd.) Kada isporučite projekat korisniku, `baza.db` se mora nalaziti u istom direktoriju kao i class fajlovi (`jar` fajl). Vaš instalacijski program se mora pobrinuti za to. Obratite pažnju da korisnik možda neće imati pravo pisanja u tom direktoriju!
- Ako URL glasi `"jdbc:sqlite::resource:baza.db"` baza se može nalaziti u `resources` folderu (kao i npr. slike, `css` fajlovi i slično) i biće uključena u `.jar` fajl, međutim biće `read-only` kada se pokreće iz `jar`-a!
- Sve datoteke koje korisnik ima pravo da mijenja bi se trebale nalaziti ispod korisničkog `home` direktorija - `System.getProperty("user.home")`. Vaš instalacijski program može iskopirati default verziju baze pod `/home/korisnik/.mojapp/mojabaza.db` i dalje URL kreirate kao:  

```
String url = "jdbc:sqlite:" + System.getProperty("user.home") +
"/.mojapp/mojabaza.db";
```

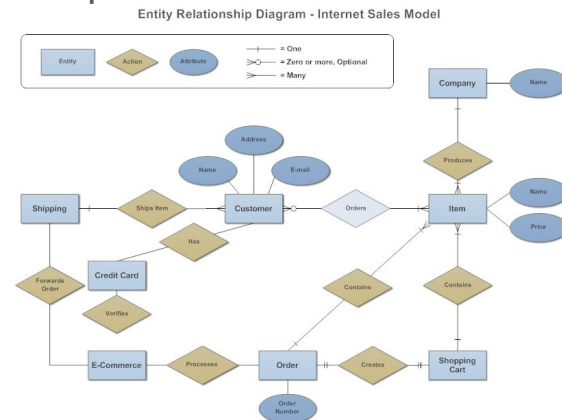
# Database tool window

- U IntelliJ Ultimate Edition ugrađen je alat za rad sa bazama pomoću kojeg možete kreirati baze, tabele itd. pod nazivom [Database tool window](#)
- Idite na **View > Tool Window > Database**. Sa desne strane će se ukazati Database kartica
- Možete samo prevući mišem (drag&drop) vašu sqlite bazu
- Ostale vrste baza kreirate klikom na dugme + ili Alt+Insert
- Ako nemate Ultimate Edition možete koristiti zasebne alate kao što je MySQL Workbench ili [SQLite Browser](#)



# Dijagram entitet-veza

- Za dizajniranje baze podataka koristi se *dijagram entitet-veza* (eng. entity-relationship diagram; ERD)
- Prije izgradnje baze podataka trebate uspostaviti kvalitetan model podataka
- Model podataka i model klasa (UML) ispunjavaju različitu svrhu, mada često mogu biti vrlo slični. Npr. ERD najčešće ne ukazuje na nasljeđivanje i sl. ERD se može automatski kreirati iz postojeće baze podataka i obrnuto. Tipovi podataka su različiti
- Oba dijagrama će se mijenjati za vrijeme životnog ciklusa razvoja softvera





# SQL injection

- Kada neki od parametara upita dolazi sa korisničkog ulaza, moguć je sigurnosni propust poznat kao *SQL injection*. Zlonamjieran korisnik može unijeti fragmente SQL upita. Primjer:

```
System.out.println("Unesite naziv knjige: ");
String knjiga = scanner.nextLine();
ResultSet result = stmt.executeQuery("SELECT * FROM biblioteka WHERE
 vlasnik=" + vlasnik + " AND naziv='" + knjiga + "'");
```

- Problem: Korisnik unese naziv knjige:

Proba' OR vlasnik='1

Upit sada glasi:

```
SELECT * FROM biblioteka WHERE vlasnik=15 AND naziv='Proba' OR vlasnik='1'
```

(cjelobrojne vrijednosti se mogu ali ne moraju staviti pod navodnike).

---

# Pripremljeni upiti

- *Pripremljeni upiti* (eng. prepared statements) osiguravaju od SQL injection, jer forsiraju određeni tip parametara (npr. vlasnik mora biti int), vrše automatski escaping navodnika u stringovima (znak ' će biti zamijenjen sa \' što je sigurno)
- Osim toga oni popravljaju performanse, jer kada se više puta izvršava isti upit, pri čemu se samo mijenja vrijednost parametra *knjiga*, taj upit se može pripremiti (optimizovati) tako da se svaki put poslije prvog izvršava brže

```
PreparedStatement ps = conn.prepareStatement("SELECT * FROM biblioteka
 WHERE vlasnik=? AND naziv=?");

ps.setInt(1, vlasnik);
ps.setString(2, knjiga);
ResultSet result = ps.executeQuery();
```

---

# Pripremljeni upiti - objašnjenje

- Objekat klase [PreparedStatement](#) dobija se metodom `prepareStatement`
- U tekstu upita umjesto parametara navodimo upitnike (nisu potrebni navodnici, oni će automatski biti ubačeni po potrebi)
- Kada želimo izvršiti upit, postavljamo vrijednost parametara koristeći metode `setInt`, `setString`, `setFloat`...
- Prvi parametar ovih metoda je redni broj (opet krećemo od 1), a drugi vrijednost koju zadajemo
- Metoda `executeQuery()` vrši upit (ako je u pitanju SELECT), a vraća **ResultSet**
- Ako je u pitanju INSERT, UPDATE ili DELETE upit, poziva se metoda `executeUpdate()`, a vraća se broj koji označava broj redova tabele koji su zahvaćeni upitom (izmijenjeni)

---

## DAL/DAO šablon dizajna (design pattern)

- *Data Access Layer* (DAL; sloj pristupa podacima) je uobičajen *arhitekturni šablon dizajna* (eng. architectural design pattern) u aplikacijama koje pristupaju bazi podataka
- Osnovna ideja je da se sve što se tiče rada sa bazom podataka - od uspostave konekcije, do konkretnih upita - drži u jednoj klasi koju nazivamo *Data Access Object* (DAO). Ova klasa se često realizuje kao *singleton* (klasa koja može imati samo jednu instancu, a konstruktor je zabranjen)
- U slučaju kompleksnijih baza podataka, to se dalje raščlanjuje na klasu za database connection i DAO klase za pojedinačne tabele u bazi, što se sve skupa drži u jednom *paketu* pod nazivom DAL
- Sastavni dio DAL šablona su DTO (*Data Transfer Object*) koji predstavljaju obične podatkovne klase (JavaBeans) u kojima će se držati podaci iz baze
- Primjer: [P20 PristupBaziDAO.zip](#)

---

# Singleton šablon dizajna

- Singleton je dobra praksa za DAO, jer za dati objekat postoji samo jedna tabela u bazi, dakle samo jedan mogući set ispravnih podataka. Singleton izbjegava da imamo više različitih pogleda na iste podatke koji mogu dovesti do konflikta
- Singleton se implementira na sljedeći način:
  - privatna referenca na vlastitu klasu
  - privatni konstruktor bez parametara (na taj način smo ustvari zabranili direktno pozivanje konstruktora) koji obavlja ostale zadatke konstruktora, npr. kreira prepared statements
  - javna metoda **getInstance** koja kreira instancu (poziva konstruktor) i dodjeljuje privatnoj referenci ako to nije urađeno prije (ako je referenca null), te vraća tu referencu

---

# Singleton šablon dizajna

```
public class NarudzbeDAO {
 private static NarudzbeDao instance = null;
 private Connection conn; /* i ostalo što treba za bazu */
 private NarudzbeDao() {
 Class.forName("..."); /* Nedostaju try catch blokovi... */
 conn = DriverManager.getConnection(...);
 }
 public static NarudzbeDao getInstance() {
 if (instance == null) instance = new NarudzbeDAO();
 return instance;
 }
 public static void removeInstance() { instance.conn.close(); instance = null; }
 ...
};
```

---

## Dobre prakse

- Prilikom kreiranja DAO objekta možemo uhvatiti izuzetak koji nastupa zato što datoteka **baza.db** ne postoji i napuniti je nekim default podacima
- Dobra praksa je da defaultne podatke za bazu držimo u SQL datoteci poznatijoj kao *database dump*. SQL datoteka se može kreirati *Export* opcijom softvera koji koristite za editovanje baze (npr. u SQLite Browser idite na File / Export / Database to SQL File...)
- SQL datoteka sadrži najobičnije SQL upite koji kreiraju tabele ("CREATE TABLE..."), a zatim ih pune podacima ("INSERT INTO..."). Ovi upiti se mogu izvršiti iz Java koda koristeći metodu **executeQuery()**
- Ovo je ujedno drugi dio odgovora na pitanje gdje držati bazu (generisat će se kod prvog pokretanja)
- Budući da je SQL datoteka praktično source code, ona se može držati na repozitoriju, te se sve promjene u default sadržaju baze mogu pratiti kao i ostale promjene u kodu

---

# Primjer SQL dump datoteke

```
BEGIN TRANSACTION;
CREATE TABLE IF NOT EXISTS `student` {
 `id` INTEGER,
 `ime` TEXT,
 `prezime` TEXT,
 `broj_indexa` INTEGER,
 PRIMARY KEY(`id`)
};
INSERT INTO `student` VALUES (1,'Student','Studenkovic',12345);
COMMIT;
```



---

# Generisanje baze

- Izuzetak izazvan činjenicom da fajl **baza.db** ne postoji se *neće* desiti prilikom uspostavljanja konekcije. Naime, SQLite driver generiše praznu bazu ako ona ne postoji. Međutim, imat ćemo izuzetak na prvom pripremljenom upitu:

```
try {
 upitSviStudenti = conn.prepareStatement("SELECT id, ime, prezime...");
} catch(SQLException e) {
 regenerisiBazu(); // Kreiramo default bazu
 try { // Ponovo pokušavamo pripremiti upit
 upitSviStudenti = conn.prepareStatement("SELECT id, ime, prezime...");
 } catch(SQLException e1) { // Ne može, znači imamo ozbiljan problem s bazom
 e1.printStackTrace();
 }
}
```

# Metoda regenerisiBazu()

```
private void regenerisiBazu() {
 Scanner ulaz = null;
 try {
 ulaz = new Scanner(new FileInputStream("baza.db.sql"));
 String sqlUpit = "";
 while (ulaz.hasNext()) {
 sqlUpit += ulaz.nextLine();
 if (sqlUpit.length() > 1 && sqlUpit.charAt(sqlUpit.length()-1) == ';') {
 try {
 Statement stmt = conn.createStatement();
 stmt.execute(sqlUpit);
 sqlUpit = "";
 } catch (SQLException e) {
 e.printStackTrace();
 }
 }
 }
 }
} // ...
```

Čitamo datoteku liniju po liniju u string

Zbog toga izvršavamo upit čim se red završava znakom ;

Ne možemo izvršiti više od jednog SQL upita

---

# Metoda regenerisiBazu()

// Nastavak metode sa prethodnog slajda

```
 ulaz.close();
 } catch (FileNotFoundException e) {
 System.out.println("Ne postoji SQL datoteka... nastavljam sa praznom bazom");
 }
}
```

---

# Metoda vratiNaDefault()

- Kod pisanja unit testova pojavljuje se problem da ako test1 promijeni stanje baze, test2 će vidjeti to promijenjeno stanje. Treba nam neki način da vratimo bazu na polazno stanje
- Najbolje rješenje je da u DAO klasi dodamo metodu koja to radi:

```
public void vratiNaDefault() throws SQLException {
 Statement stmt = conn.createStatement();
 stmt.executeUpdate("DELETE FROM student");
 regenerisiBazu();
}
```

- Naime, podsjetimo se da u SQL dump-u kod kreiranja tabele stoji **CREATE TABLE IF NOT EXISTS...** Znači, kod iznad neće ponovo kreirati tabele, ali će ih napuniti default podacima koje smo stavili u dump (Student Studenković)

---

# DAL i MVC

- MVC model dat ranije se može vrlo lako spojiti sa bazom podataka
- Naše Model klase sada postaju DAO klase!
- Primjer: [P20\\_BibliotekaDB.zip](#)

---

## Dodavanje baze u MVC

- Naša Model klasa će sada postati DAO (Data Access Object)
- Postaće singleton klasa (generalno nema smisla imati više instanci baze)
  - Pošto je Model sada singleton, u Main klasi umjesto  
`NestoModel model = new NestoModel();`  
sada imamo:  
`NestoModel model = NestoModel.dajInstancu();`
- Konstruktor klase treba učitati podatke iz baze u attribute
- Pošto je atribut tipa **ObservableList**<Nesto> moramo napraviti upit koji učitava podatke iz baze u listu - primjer na sljedećem slajdu

# Konstruktor model klase

Konstruktor je privatn (singleton klasa)

```
private BibliotekaModel() {
 try {
```

Vršimo upit kojim preuzimamo sve knjige

```
 conn = DriverManager.getConnection("jdbc:sqlite:resources/baza.db");
 stmt = conn.prepareStatement("SELECT autor, naslov, isbn, brojstran
```

Za svaki rezultat kreiramo instancu klase Knjiga...

... i dodajemo u listu

```
 ResultSet rs = stmt.executeQuery();
```

```
 while (rs.next()) {
```

```
 Knjiga k = new Knjiga(rs.getString(1), rs.getString(2), rs.getString(3), rs.getInt(4));
 knjige.add(k);
```

```
 if (trenutnaKnjiga == null) trenutnaKnjiga = new SimpleObjectProperty<Knjiga>(k);
```

Ako nije bilo knjiga u bazi, trenutnaKnjiga ne smije biti null referenca

```
 catch (SQLException e) {
```

```
 System.out.println("Neuspješno čitanje iz baze: " + e.getMessage());
```

Prvi rezultat će biti trenutna knjiga

```
 }
```

```
 if (trenutnaKnjiga == null) trenutnaKnjiga = new SimpleObjectProperty<Knjiga>();
```

```
 }
```

Sve operacije nad bazom bacaju SQLException

---

# Izmjena podataka

- Operacija za dodavanje knjige bi sada trebala izvršiti INSERT upit
- Promjenu i brisanje knjige moramo uhvatiti pomoću listenera i izvršiti UPDATE i DELETE upite
- Radi jasnog razdvajanja djelatnosti (separation of concerns), Model klasa ne bi trebala sadržavati listenere, Controller klasa ne bi trebala sadržavati SQL upite
- Zbog toga dodajemo u **NestoModel** klasu metode: **dodajNesto**, **promijeniNesto**, **obrisiNesto**. Ove metode ćemo pozivati iz akcija i listenera u controller klasi



---

## Dodavanje GUI na gotovu DAL aplikaciju

- Analogno, ako imamo napravljenu aplikaciju po DAL šablonu, možemo na nju dodati JavaFX GUI i transformisati je u MVC aplikaciju
- U DTO klasama svi atributi trebaju sada biti *properties*
- DAO klasa postaje model
  - Zbog prikaza kolekcijskih grafičkih elemenata (**ListView**, **ChoiceBox**...) možda ćemo u privatne attribute morati dodati **ObservableList**, ali onda se moramo pobrinuti da sve metode koje mijenjaju bazu uredno ažuriraju ove liste!
- Controller klasu implementiramo kao i ranije

---

# **Razvoj programskih rješenja**

Mrežno programiranje

---

# URL

- Svaki resurs na webu se označava jedinstvenom adresom koju nazivamo *univerzalni lokator resursa* (eng. Universal Resource Locator – URL)
- Pored toga, spominje se i *univerzalni identifikator resursa* (URI) koji može biti URL ili ime resursa (URN). Primjer URL:

`https://zamger.etf.unsa.ba/index.php?sta=izvjestaj/  
predmet&predmet=9&ag=14&...`

- URL se sastoji od: mrežnog **protokola** ili šeme pomoću kojeg se pristupa resursu (http, https, ftp...), znaka dvotačka, autoriteta, putanje, upita i fragmenta

---

# URL

<https://zamger.etf.unsa.ba/index.php?sta=izvjestaj/predmet&predmet=9&ag=14&...>

URL se sastoji od sljedećih dijelova, od čega su samo prva dva obavezni:

- mrežnog **protokola** ili **šeme** pomoću kojeg se pristupa resursu (http, https, ftp...)
- znaka dvotačka
- oznake **autoriteta** (eng. authority) koja predstavlja domensko ime servera, a počinje znakovima //
- **putanje** (puta, eng. path) do resursa, koja može (ali ne mora) počinjati znakom /
- **upita** (eng. query) koji predstavlja podatke koji se šalju kao parametri web lokaciji, pri čemu su podaci razdvojeni znakom &, a u formatu su ključ=vrijednost; upit počinje znakom ?
- **fragment** koji počinje znakom # a predstavlja najčešće informacije za klijenta koje nisu relevantne serveru (ali ih on svejedno dobija)

---

# Klasa URL

- Klasa [URL](#) omogućava da se provjeri da li string predstavlja validan URL, te nudi metode za pristup pojedinačnim komponentama
- U slučaju da URL nije validan, baca se izuzetak **MalformedURLException**:

```
Scanner scanner = new Scanner(System.in);
String adresa = scanner.nextLine();
try {
 URL url = new URL(adresa);
 System.out.println("URL je ok, protokol " + url.getProtocol() +
 ", autoritet " + url.getAuthority());
} catch (MalformedURLException e) {
 System.out.println("String "+adresa+" ne predstavlja validan URL");
}
```

---

# Preuzimanje podataka s Interneta

- Pored toga klasa URL posjeduje metodu **openStream()** koja otvara InputStream prema datom resursu. Dalje s ovim streamom možete raditi sve što poželite
- Razne web bazirane aplikacije vam nude web bazirani API pomoću kojeg vaš program može preuzeti neke podatke ili obaviti neki zadatak putem API-ja
- Npr. ako API daje podatke u XML formatu (SOAP) možete koristiti XML parser. Primjer: <http://www.geoplugin.net/xml.gp?ip=217.75.199.45>

Primjer: [P18\\_URL.zip](#)

## Primjer korištenja metode `openStream()`

- Sljedeći kod će pristupiti URLu <http://www.etf.unsa.ba/> i preuzeti sadržaj ove web stranice u string:

```
URL url = new URL("http://www.etf.unsa.ba");
BufferedReader ulaz = new BufferedReader(new InputStreamReader(
 url.openStream(), StandardCharsets.UTF_8));
String sadrzaj = "", line = null;
while ((line = ulaz.readLine()) != null)
 sadrzaj = sadrzaj + line;
```

- Nedostaci: jedina podržana metoda je GET, ne možemo koristiti PUT, POST, DELETE, ne možemo definisati header polja, ne možemo poslati request body...

---

# JSON

- Pored XMLa u raširenoj upotrebi za web API-je su formati JSON i YAML. Pošto je JSON najpopularniji fokusiraćemo se na njega
- JSON (JavaScript Object Notation) nastao je kao pokušaj da se objekti u programskom jeziku JavaScript zapišu u tekstualnu datoteku u formatu samog jezika. Drugim riječima, JSON možete doslovce kopirati u JavaScript program i dobićete istovjetan objekat. Za druge programske jezike postoje biblioteke za parsiranje
- Kod JSON formata vitičaste zagrade označavaju objekat, a uglaste niz. Unutar toga se mogu nalaziti drugi objekti/nizovi ili atributi u formatu "ključ":"vrijednost". Svi članovi objekta su razdvojeni zarezom. Članovi ne moraju biti imenovani (objekti, nizovi itd. ne moraju imati ime). Na vrhu hijerarhije se nalazi jedan bezimени objekat



---

# JSON primjer

- XML datoteka sa državom u JSON formatu bi izgledala ovako:

```
{
 "stanovnika": 3500000,
 "naziv": "Bosna i Hercegovina",
 "glavnigrad": {
 "stanovnika": "400000",
 "naziv": "Sarajevo"
 },
 "povrsina": {
 "jedinica": "km2",
 "vrijednost": "51129"
 },
 "moneta": "KM"
}
```

---

# JSON biblioteke

- Postoji više JSON biblioteka za Java jezik. Najpopularnije su [GSON](#), [org.json](#) i [Jackson](#). Zbog jednostavnosti ovdje ćemo koristiti org.json. Da biste uključili biblioteku u projekat, trebate koristiti Project Structure > Libraries > + > From Maven... i ukucati sljedeće Maven "koordinate": **org.json:json:20201115**
- Klasa **JSONObject** ima konstruktor koji prima string. Primjer:

```
String json = "{ \"kljuc\": \"vrijednost\" }";
JSONObject jsonObject = new JSONObject(json);
System.out.println(jsonObject.getString(\"kljuc\")); // ispisace: vrijednost
```

---

# Rad sa JSONom

Klasa **JSONObject** posjeduje metode za pristup elementima kao što su:

- **getString**(atribut) - daje vrijednost atributa u formi stringa
- **getInt, getDouble...** - konvertuje vrijednosti različitih tipova
- **getJSONObject**(ime) - drugi objekat sadržan u prvom
- **getJSONArray**(ime) - niz

```
int stanovnika = obj.getInt("stanovnika");
String glavniGrad = obj.getJSONObject("glavnigrad").getString("naziv");
int glavniStanovnika = obj.getJSONObject("glavnigrad").getInt("stanovnika");
...
```

Biće izvršena  
konverzija tipa

---

# Pristup REST JSON web servisu

- Način rada kao do sada podržava samo trivijalne servise za koje je pristupna metoda GET i svi parametri su dio URLa:

```
URL url = new URL("https://aws.random.cat/meow");
BufferedReader ulaz = new BufferedReader(new InputStreamReader(
 url.openStream(), StandardCharsets.UTF_8));
String json = "", line = null;
while ((line = ulaz.readLine()) != null)
 json = json + line;
JSONObject jsonObject = new JSONObject(json);
...
```

---

# URLConnection

- Za složenije API-je koristimo klasu **URLConnection** koja nudi više opcija. Potrebno je da pozovemo metodu **openConnection()** klase URL:

```
URL url = new URL("https://aws.random.cat/meow");
URLConnection con = (URLConnection)url.openConnection();
```
- Postavljanje HTTP metode vršimo funkcijom **setRequestMethod**:

```
con.setRequestMethod("POST");
```
- Postavljanje polja zaglavlja vršimo metodom **setRequestProperty**. Za API-je koji koriste JSON u pravilu treba postaviti polja Content-Type i Accept na vrijednost application/json:

```
con.setRequestProperty("Content-Type", "application/json; utf-8");
con.setRequestProperty("Accept", "application/json");
```

---

# URLConnection

- Kod vrlo jednostavnih API-ja parametri se šalju u sklopu samog URLa (kroz "upit"). Međutim, polje za upit nije dovoljno fleksibilno i ograničena je njegova veličina. Mnogi REST API-ji zahtijevaju da se vrijednost pošalje u tijelu zahtjeva (request body) i da budu JSON kodirani. Ovo postizemo tako što najprije metodom **setDoOutput(true)** označavamo da je komunikacija dvosmjerna, a zatim šaljemo tekst kroz **OutputStream**:

```
con.setDoOutput(true);
String parametri = "{\"ime\": \"Hamo\", \"prezime\": \"Hamic\"}";
OutputStream izlaz = con.getOutputStream();
byte[] input = jsonString.getBytes("utf-8");
os.write(input, 0, input.length);
```

---

# URLConnection

- Konačno, čitanje sadržaja koji je vratio server se radi slično samo što koristimo metodu **getInputStream** klase **URLConnection**:

```
BufferedReader ulaz = new BufferedReader(new
InputStreamReader(con.getInputStream(),
 StandardCharsets.UTF_8));
String json = "", line = null;
while ((line = ulaz.readLine()) != null)
 json = json + line;
JSONObject jsonObject = new JSONObject(json);
...
```

---

# Klijent-server komunikacija

- Java aplikacije se mogu koristiti za klasičnu mrežnu komunikaciju po vlastitom protokolu, pri čemu vaša aplikacija može igrati ulogu klijenta ili servera. Ovo se postiže klasom **Socket**.

Primjer klijenta:

```
String adresa = "192.168.1.8";
int port = 2345;
try {
 Socket konekcija = new Socket(adresa, port);
 InputStream ulaz = konekcija.getInputStream();
 OutputStream izlaz = konekcija.getOutputStream();
 ...
 konekcija.close();
} catch (IOException e) {
 ...
}
```



---

# Klijent-server komunikacija

Primjer servera:

```
int port = 2345;
try {
 ServerSocket server = new ServerSocket(port);
 while (true) {
 Socket konekcija = server.accept();
 InputStream ulaz = konekcija.getInputStream();
 OutputStream izlaz = konekcija.getOutputStream();
 ...
 konekcija.close();
 }
} catch (IOException e) {
 ...
}
```

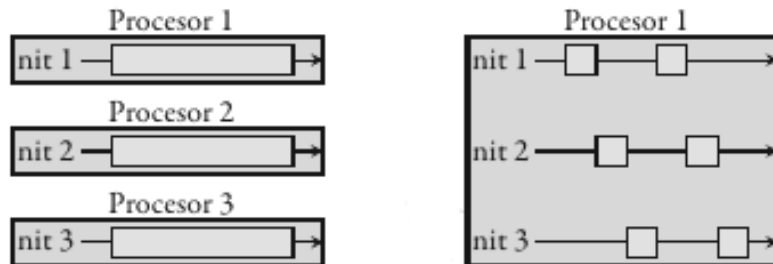
---

# **Razvoj programskih rješenja**

Višenitno programiranje

# Istovremeno (paralelno) izvršenje

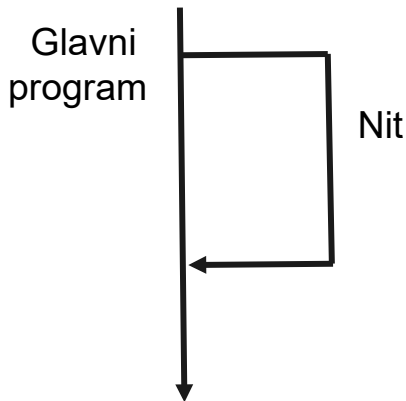
- Računari mogu izvršavati više zadataka istovremeno
- Ako računar ima više procesora ili jezgri, može se izvršavati toliko istovremenih zadataka
- Ako ima samo jedan procesor/jezgru ili su svi zauzeti, procesor se vrlo brzo prebacuje između aktivnih zadataka tako da se stvara privid istovremenosti



---

# Niti (threads)

- Niti (eng. threads) su mehanizam pomoću kojeg možete postići da se nekoliko zadataka izvršava istovremeno i time uposliti sve jezgre koje vaš računar ima
- Sam program koji pokrećete predstavlja jednu (glavnu) nit. Iz nje možete pokretati druge niti
- JavaFX interfejs se izvršava u drugoj niti kako se ne bi desilo da se ekran prestane osvježavati dok se izvršava neki zadatak u pozadini



---

# Kreiranje nove niti

- Da biste kreirali novu nit u programu, potrebno je da imate klasu *zadatak* (eng. task) koja predstavlja neki posao koji treba obaviti u pozadini (paralelno)
- Ta klasa treba da implementira interfejs **Runnable**
- Ovaj interfejs zahtijeva samo jednu metodu: **void run()** koja sadrži kôd koji želite da se izvršava

```
class Brojevi implements Runnable {
 @Override
 public void run() {
 for (int i=1; i<=100; i++)
 System.out.print(" " + i);
 }
}
```

# Pokretanje niti

- Ako bismo sada pozvali metodu **run()**, ova nit se ne bi izvršavala u pozadini, jer je to metoda kao i svaka druga
- Umjesto toga moramo kreirati objekat klase **Thread** i pozvati njegovu metodu

**start :**

```
Brojevi brojevi = new Brojevi();
Thread thread = new Thread(brojevi);
thread.start();
```

- Klasa **Thread** nam nudi razne korisne metode za upravljanje nitima

Primjer: [P19\\_Threads.zip](#)

# Koordinacija niti

- Statičku metodu **yield()** možemo pozvati kad god želimo da trenutna nit prepusti kontrolu nekoj drugoj niti da se izvršava
- Metodu **sleep(long milis)** možemo pozvati da pauziramo trenutnu nit za određeni broj milisekundi, tokom kojih se izvršavaju druge niti
- Primjećujemo da se poruka "Program završen" ispisala prije nego što su se niti završile. Metoda **join()** zaustavlja trenutnu nit dok se neka druga nit ne završi

```
try {
 thread1.join();
 thread2.join();
 thread3.join();
} catch (InterruptedException e) {
 e.printStackTrace();
}
System.out.println("Program završen");
```

---

# Lambda funkcije

- Pravljenje pomoćnih klasa iziskuje dosta vremena. Koristeći lambda sintaksu možemo na licu mjesta definisati našu funkciju:

```
Thread thread = new Thread(() -> {
 for (int i=1; i<=100; i++)
 System.out.print(" " + i);
});
```



# Executor klasa

- Situacija u kojoj želimo da pokrenemo veći broj niti (često istog koda) je dosta česta. Tu se koristi klasa **Executor** koja nam olakšava ovaj zadatak. Primjer:

```
ExecutorService executor = Executors.newFixedThreadPool(3);
executor.execute(() -> {
 for (int i=1; i<=100; i++) {
 System.out.print(" " + i);
 }
});
...
executor.shutdown();
```

---

# Niti i JavaFX

- Ako želimo da napravimo nit koja mijenja nešto na grafičkom interfejsu, ta nit mora biti povezana sa JavaFX thread-om (rekli smo da je to zasebna nit u odnosu na glavnu)
- To postižemo pozivom **Platform.runLater()** koja prima **Runnable** odnosno ponovo može primiti lambda funkciju
- Primjer: [P19\\_Blinky.zip](#) - u ovom primjeru pravili smo labelu koja mijenja sadržaj periodično koristeći pomoćnu nit
- Bolje rješenje za ovo je koristiti JavaFX animacije (o kojima nismo pričali ove godine...)

---

# Validacija putem mreže

```
new Thread(() -> {
 try {
 URL url = new URL(adresa);
 BufferedReader ulaz = new BufferedReader(new
 InputStreamReader(url.openStream(), StandardCharsets.UTF_8));
 String json = "", line = null;
 while ((line = ulaz.readLine()) != null)
 json = json + line;
 if (...) {
 Platform.runLater(() -> { polje.setStyle("nije-ispravno"); });
 } catch(...)
 }
}).start();
```

---

## Stanje trke (race condition)

- Stanje trke (eng. race condition) nastaje kada dvije niti pokušavaju istovremeno da promijene istu vrijednost - "utrkuju se" ko će prije promijeniti vrijednost
- Npr. recimo da dvije niti pokušavaju da izvrše naredbu:  
`brojac = brojac + 1;`  
pri čemu brojac nije lokalna promjenljiva za nit
- Kao što znamo, najprije se uzima vrijednost promjenljive brojac, na nju se dodaje 1, a zatim se ta vrijednost upisuje u promjenljivu brojac. Dok jedna nit izračunava brojac+1 druga nit je možda već izračunala i upisala tu vrijednost tako da prva nit sada ima zastarjelu vrijednost i neće uspješno obaviti posao
- Primjer kompletne klase sa ilustracijom izvršenja u literaturi Živković

---

# Sinhronizacija niti

- Stanje trke se sprječava tako što se u kodu niti označava određeni dio koji nazivamo "kritično područje" te se zabranjuje da dvije niti istovremeno izvršavaju kritično područje
- Mehanizam kojim se ovo obično postiže je ključ (eng. lock). Kada jedna nit uzme ključ, ostale niti moraju čekati da ta nit vrati ključ da bi prošle odgovarajuće mjesto u kodu
- U programskom jeziku Java uvedena je ključna riječ **synchronized**. Metodu označenu ovom ključnom riječi može izvršavati samo jedna nit u datom trenutku. Npr. u našoj klasi možemo imati:

```
public synchronized void uvecajBrojac() {
 brojac = brojac + 1;
}
```

---

# Sinhronizovani izrazi

- Drugi način sinhronizacije u Javi je da označimo neki izraz kao **synchronized**. Na taj način neki objekat ili vrijednost izraza može se koristiti kao "ključ" i koristiti na više mjesta u kodu
- Primjer:

```
synchronized (b) {
 if (b.value() == 0)
 doSomething();
}
```

- Druga nit neće moći izvršavati niti jedan blok označen sa "**synchronized (b)**" dok se ovaj blok ne završi
- Kada nit pokušava pristupiti resursu možemo zamisliti da ona u nekoj beskonačnoj petlji provjerava da li je resurs slobodan, ali ustvari nit je u stanju spavanja (sleep) tako da ne opterećuje procesor

# Uzajamno zaključavanje (deadlock)

- Sinhronizacija i korištenje ključeva uvode novi potencijalni problem. Dešava se da nit A koristi resurs 1, a želi da pristupi resursu 2, dok nit B koristi resurs 2, a želi da pristupi resursu 1. Pošto se u ovom slučaju neće nikada osloboditi ni jedan ni drugi resurs, obje niti su blokirane.
- Ovakvu situaciju nazivamo deadlock (uzajamno zaključavanje)
- Jedino rješenje za deadlock je reorganizacija koda i resursa

Nit 1:

```
synchronized (r1) {
 ...
 synchronized (r2) {
 ...
 }
}
```

Nit 2:

```
synchronized (r2) {
 ...
 synchronized (r1) {
 ...
 }
}
```

---

## Komunikacija između niti

- Nekada je potrebno da iz jedne niti obavijestimo drugu da je npr. neki resurs spreman
- Nit može poslati poruku da je resurs spreman pozivom metode **notify()** nad resursom. Ova metoda je definisana u klasi **Object** što znači da postoji u svim Java klasama. Druga nit čeka na resurs metodom **wait()**
- Može se desiti da više niti čeka na isti resurs. **notify()** će "osloboditi" samo jednu od njih. Ako hoćemo da oslobodimo sve niti pozivamo **notifyAll()**
- Slično preko klase **ReentrantLock** (Liang)



---

# Sinhronizovane kolekcije

- Kolekcije opisane na ranijim predavanjima nisu thread-safe, što znači da bilo koja operacija nad kolekcijom (čak i različitim elementima) može izazvati *race condition*
- Da ne biste morali zaključavati čitavu kolekciju, možete koristiti sinhronizovane kolekcije

- Sinhronizovana varijanta kolekcije se dobija pozivom

**`Collections.synchronizedCollection`** npr.

```
Collection<String> lista = Collections.synchronizedCollection(
 new ArrayList<String>());
```

- U slučaju da jedna nit dodaje/briše elemente iz kolekcije, kretanje iteratorom iz druge niti se ne može tek tako sinhronizovati. U tim slučajevima, iterator sinhronizovanih kolekcije baca **`ConcurrentModificationException`**

---

# **Razvoj programskih rješenja**

Izvještavanje

---

# Zašto izvještavanje?

- Česti problemi u poslovnim aplikacijama:
  - Generisati dokument iz baze podataka koji lijepo izgleda na ekranu i printeru (PDF) - npr. faktura sa memorandumom firme, grafikoni i slično
  - Odštampati dokument iz aplikacije
  - Izvoz u Word, Excel, PDF, HTML i druge formate
- Sve ovo možete ručno, ali bi vam oduzelo mogo vremena

---

# Business Intelligence

- Termin koji se kod nas često prevodi kao "poslovna inteligencija" - što je zbunjujuće, jer najčešće ne uključuje nikakvu vještačku inteligenciju
- Umjesto toga preferirani prevod je "poslovno izvještavanje"
- Ovaj termin obuhvata prikupljanje i obradu podataka vezanih za poslovanje firme, njihovo prezentovanje i vizualizaciju sa ciljem donošenja bitnih poslovnih odluka
- Često korišteni pristupi su: Data mining, Data warehousing, OLAP. Naglasak je na vizualizaciji podataka, interaktivnosti

---

# Alati za izvještavanje

- **Crystal Reports** - komercijalni alat koji se dobro integrisao sa VisualStuđiom, bio je uključen u VS 2003 i 2008. Piratske verzije kruže.
- **Oracle Reports**
- **Microsoft SQL Server Reporting Service** - dolazi uz SQL server. **MS VisualStudio BI** (Business Intelligence), BIDS (BI Designer Service), SSDT (SQL Server Data Tools)...
- **Jasper Reports** - open source rješenje pravljeno u Java jeziku

U nastavku ćemo raditi sa Jasper-om

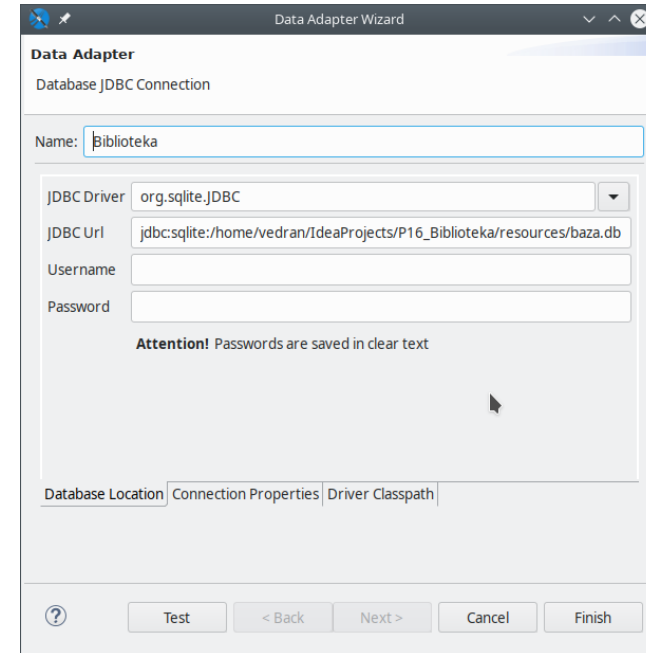
---

# Jaspersoft Studio

- Prvi korak ka dodavanju izvještaja u vaš projekat je da downloadujete i instalirate program pod imenom [Jaspersoft Studio](#) koji služi za kreiranje izvještaja
- Izvještaji su u JRXML formatu (vrsta XMLa) i mogu se kreirati i ručno, ali je puno lakše kroz grafički alat
- Primjer: [P21\\_BibliotekaJasper.zip](#)

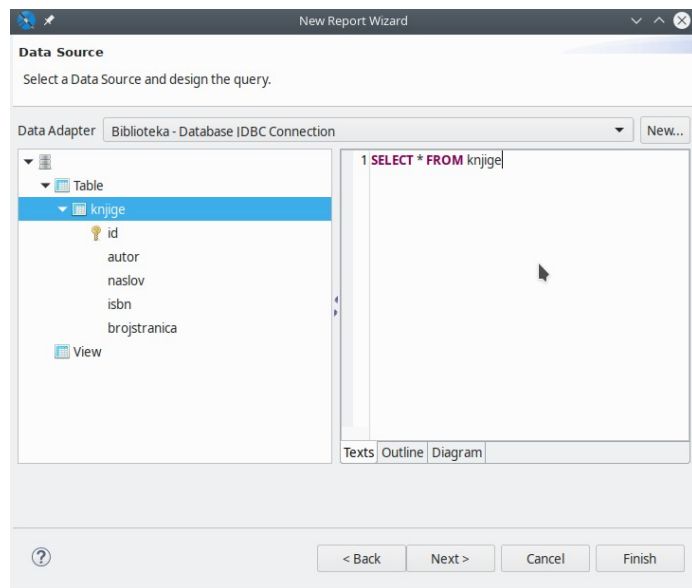
# Data Adapter

- Kada pokrenete Studio najprije je potrebno da kreirate "Data Adapter" (data source) za pristup vašoj bazi iz koje će se povlačiti podaci za izvještaj.
- Idite na **File > New > Data Adapter** ili kliknite na ikonicu.
- Izaberite **Database JDBC Connection** mada možete kreirati adaptere i za nekoliko vrsta datoteka ili neke druge izvore (npr. Hibernate sesiju, OLAP i slično)
- Pod **Name** unesite neko opisno ime
- Pod **JDBC Driver** izaberite vrstu baze podataka npr. SQLite, MySQL, Oracle i slično
- Ako radite sa SQLite morate ručno u **JDBC Url** otkucati kompletan apsolutni put do db datoteke



# Novi izvještaj

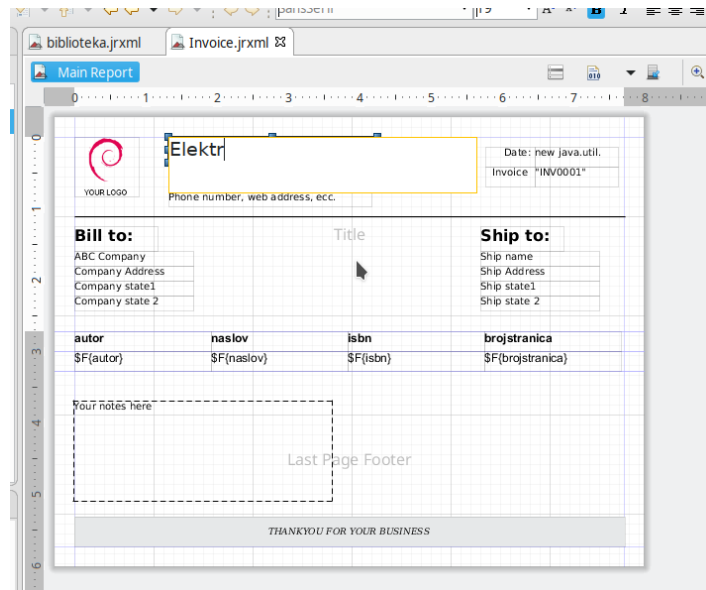
- Sada kliknite na **File > New > Jasper Report**
- Možete izabrati jedan od ponuđenih template-a npr. Invoice za fakturu. Kliknite **Next>**
- Izaberite naziv datoteke za izvještaj npr. biblioteka.jrxml. Kliknite **Next>**
- Izaberite Data Adapter koji ste maloprije kreirali. Sada u lijevom dijelu prozora možete vidjeti šemu baze, a u desnom dijelu prozora se nudi da kreirate upite koji dobavljaju podatke koji vam trebaju za izvještaj. Možete otkucati upit (**Text**) ili ga kreirati kroz selekciju (**Outline**) ili crtanjem (**Diagram**)





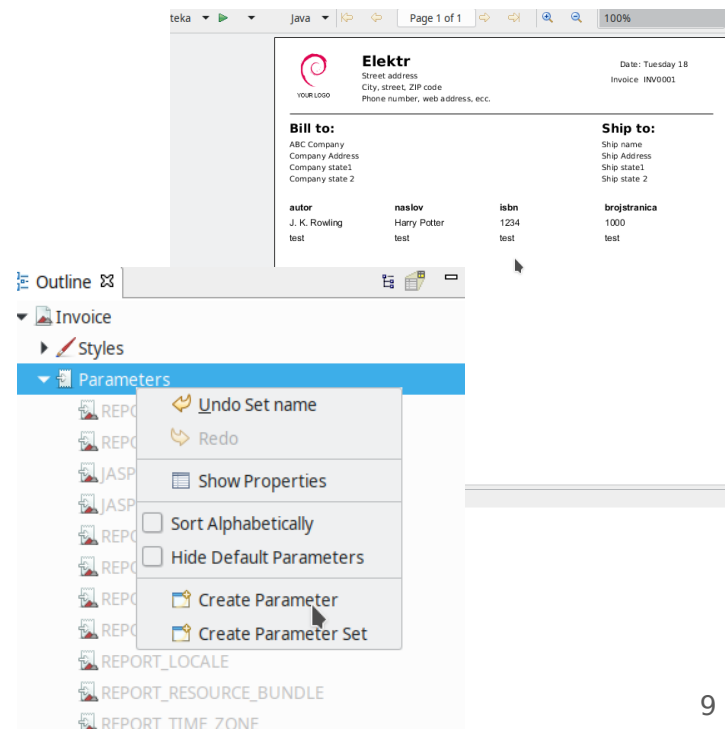
# Izbor polja

- U sljedećem koraku birate koje kolone vašeg upita predstavljaju polja koja utiču na izvještaj, a zatim možete da grupišete po određenim poljima (kao dodatni Group By podupit)
- Sada je vaš izvještaj kreiran i možete da crtate po njemu, kao što biste radili u nekom grafičkom programu
- Polja upita prepoznajete po tome što počinju sa \$F - \$F{autor} je kolona upita "autor". Redovi će se automatski dodavati u tabelu



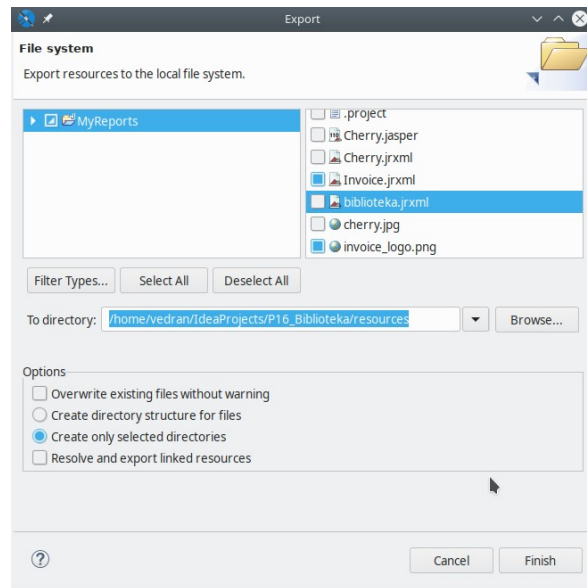
# Preview i parametri

- Klikom na karticu Preview možete vidjeti kako izvještaj izgleda sa aktuelnim podacima iz baze. Preview je read-only
- Izvještaj može primati *parametre* koji ne dolaze iz baze (upita) nego iz Java koda koji poziva izvještaj. U donjem lijevom uglu ekrana kliknite na **Parameters > Create Parameter**. U donjem desnom uglu možete izabrati tip parametra npr. **java.lang.String**



# Dodavanje izvještaja u projekat

- Najprije je potrebno da iz Studia izvezemo .jrxml datoteku projekat. Najlakši način je da izaberete **File > Export Files To...** a zatim u sekciji Browse pronađete apsolutni put do **resources** foldera unutar vašeg projekta
- Predlažemo da unutar resources kreirate **reports** pa da u taj direktorij zapišete datoteke
- Ako šablon (template) uključuje i neke ilustracije npr. invoice\_logo.png koje želite dodati u vaš projekat, ne zaboravite da ih označite na spisku u gornjem desnom uglu



---

# JasperReports Library

- Da bi se izvještaj mogao vidjeti iz vaše Java aplikacije, potrebno je da u nju ugradite preglednik izvještaja (JasperReports Viewer). Ovo je vrlo jednostavno i šablonski
- Potrebno je da u projekat ugradite Jasper biblioteku (JasperReports Library)
- Idite na **File > Project Structure > Libraries > From Maven...**, izaberite Maven koordinate **net.sf.jasperreports:jasperreports:6.12.0**
- Sljedeći korak koji je potreban je da kreirate PrintReport klasu (kod je šablonski i slijedi na sljedećem slajdu)
- U kodu promijenite naziv jrxml datoteke ili, ako imate više izvještaja, to može biti jedan od parametara metode

---

```
public class PrintReport extends JFrame {
 public void showReport(Connection conn) throws JRException {
 String reportSrcFile = getClass().getResource("/reports/biblioteka.jrxml").getFile();
 String reportsDir = getClass().getResource("/reports/").getFile();

 JasperReport jasperReport = JasperCompileManager.compileReport(reportSrcFile);
 // Fields for resources path
 HashMap<String, Object> parameters = new HashMap<String, Object>();
 parameters.put("reportsDirPath", reportsDir);

 ArrayList<HashMap<String, Object>> list = new ArrayList<HashMap<String, Object>>();
 list.add(parameters);

 JasperPrint print = JasperFillManager.fillReport(jasperReport, parameters, conn);
 JasperViewer viewer = new JasperViewer(print, false);
 viewer.setVisible(true);
 }
}
```

Put do /reports/  
foldera šaljem  
kao parametar  
izveštaju

Metoda fillReport  
prima konekciju  
na bazu

---

## Poziv klase PrintReport

- Uočavamo da metoda showReport prima Connection objekat. Moramo u DAO klasu (model klasa) dodati metodu koja vraća privatni atribut **conn**
- Sada na formu dodamo neko dugme Štampaj knjige čiji kod izgleda ovako (naravno trebamo odlučiti šta raditi u slučaju JRException):

```
public void stampajKnjige(ActionEvent actionEvent) {
 try {
 new PrintReport().showReport(model.getConn());
 } catch (JRException e1) {
 e1.printStackTrace();
 }
}
```

---

# Pristup resursima

- Obratite pažnju da prethodni kod klase PrintReport čita jrxml datoteku iz resursa. Ako se u jrxml šablonu pristupa drugim datotekama (npr. slikama), ovo neće raditi, jer se one nalaze u resursima. Neće raditi ni apsolutni ni relativni put
- Potrebno je da put do resources foldera proslijedite kao parametar izvještaju (u Java kodu već urađeno)

- U JRXML kodu trebate deklarirati parametar:

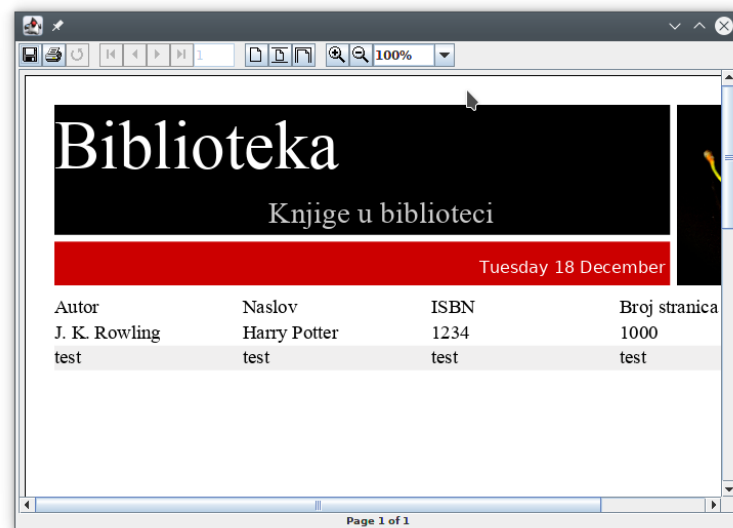
```
<parameter name = "reportsDirPath" class = "java.lang.String"/>
```

- Zatim slici pristupa kreirajući File opjekat iz parametra i relativnog puta:

```
<imageExpression class="java.io.File"><![CDATA[new
File(${reportsDirPath}, "cherry.jpg")]]></imageExpression>
```

# ReportViewer

- Za ovaj Report Viewer dokumentacija kaže da je "demo viewer", tako da bi poželjno bilo da ga prilagodite tj. da napravite svoj
- Obratite pažnju da je pravljen u Swing-u tako da prilično ružno izgleda u našoj JavaFX aplikaciji, pa ga možemo i ljepše napraviti
- Save opcija nudi formate: PDF, ODT, DOCX, XSLX, HTML... Print opcija nudi vaše systemske printere
- Sve mogućnosti viewera dostupne su i kroz Java kod, pa možete implementirati opciju Print bez pregleda





---

# Report Server

- Kod velikih poslovnih aplikacija, a pogotovo kod web-baziranih aplikacija, uobičajeno je da se posao generisanja izvještaja povjeri posebnom serveru koji se naziva Report Server
- Moguće je instalirati [JasperReports Server](#) na Windows, Linux ili MacOSX
- U Jasper Studio postoji opcija "Publish to server" kojom postavljamo izvještaj na server
- Izvještaju se pristupa putem URLa. Klikom na URL korisnik u svom web pregledniku otvara web bazirani preglednik sa sličnim mogućnostima kao report viewer. Parametri se proslijeđuju putem URLa

---

# **Razvoj programskih rješenja**

Internacionalizacija

---

## i18n i l10n

- *Lokalizacija* (eng. localization, **l10n**) je prilagođavanje nekog proizvoda (ne nužno softverskog) za neko konkretno lokalno tržište npr. neke države
- *Internacionalizacija* (eng. internationalization, **i18n**) je dizajn i razvoj proizvoda na način koji olakšava kasniju lokalizaciju
- Primjeri:
  - Ako nudite aplikaciju za međunarodno tržište, mora biti dostupna na stranim jezicima
  - Aplikacije za državne institucije BiH moraju biti dostupne na bosanskom, hrvatskom i srpskom jeziku, a često i engleskom

---

# Šta se sve lokalizira?

- Sve poruke (labele) u korisničkom interfejsu trebaju biti prikazane na odabranom jeziku
- Dokumentacija (Help itd.) treba koristiti ilustracije (screenshotove) sa istim jezikom
- Format datuma u engleskom govornom području je mjesec/dan/godina, a kod nas je dan.mjesec.godina, mada ima i država gdje je godina/mjesec/dan itd.
- U nekim zemljama je uobičajen AM/PM format za vrijeme, a u nekim 24-satni format
- Mjerne jedinice u SAD su: inch, foot, yard, pound, ounce... u ostatku svijeta cm, m, km, kg, l...
- Valuta treba biti prilagođena državi (u finansijskim aplikacijama)
- U jezicima gdje se piše zdesna nalijevo (npr. arapski) običaj je da su i meniji poredani s desne strane, ikone u toolbaru itd.

---

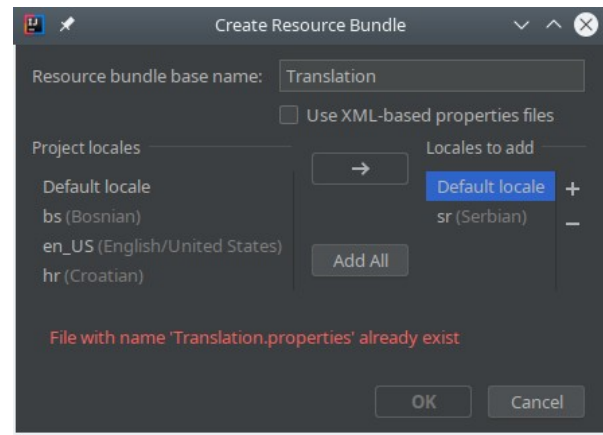
# Koji jezik koristiti?

- Korisničko okruženje (operativni sistem, web preglednik) definiše default jezik koji korisnik želi da koristi i koji je odabrao prilikom instalacije operativnog sistema
- Ipak, trebalo bi da postoji mogućnost promjene jezika
- Java klasa **Locale** služi za manipulaciju lokalnim (regionalnim) postavkama u koje spadaju država, jezik, i sl. Npr. da dobijete kod jezika možete pisati:  
`Locale.getDefault().getLanguage()`
- Standard **ISO 639** definiše kodove jezika, a **ISO 3166** kodove država. Obratite pažnju da se oni često razlikuju, npr. **ba** je kod za Bosnu i Hercegovinu, a **bs** za bosanski jezik
- Nekada je dovoljno kao lokal zadati jezik, a nekada se treba kombinovati jezik i država, sa znakom underscore i država velikim slovima: "en\_US", "bs\_BA", "de\_DE".

# ResourceBundle

Prvi korak ka internacionalizaciji je kreiranje ResourceBundle:

- Kreirajte **resources** folder kao i ranije (ako nije kreiran)
- Desnim klikom na ovaj folder izaberite **New > Resource Bundle**
- Otvoriće se prozor kao na slici desno. Izaberite naziv za bundle fajl (npr. Translation - ako imate kompleksan projekat možda je zgodno razdvojiti ga na više bundle-ova)
- Pod Locales dodajte jezike koje za sada želite imati u vašoj aplikaciji. [Kodovi za jezike](#) su: en\_US (engleski SAD), bs (bosanski), hr (hrvatski), sr (srpski) itd.



---

# Učitavanje Resource Bundle

- Potrebno je promijeniti dio koda gdje se otvara fxml datoteka i kreira scena (vjerovatno Main)

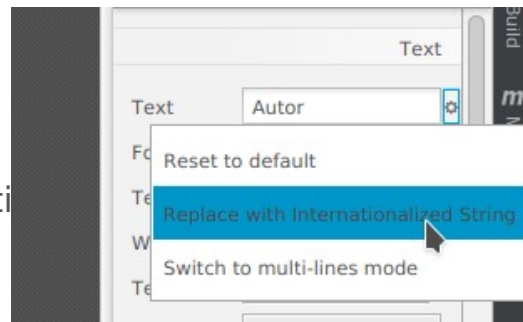
- Sada taj kod treba glasiti:

```
ResourceBundle bundle = ResourceBundle.getBundle("Translation");
FXMLLoader loader = new FXMLLoader(getClass().getResource(
 "/biblioteka.fxml"), bundle);
```

- Uočavamo da konstruktor klase FXMLLoader sada prima i drugi parametar, a to je ResourceBundle

# Lokalizacija korisničkog interfejsa

- U Scene Builderu selektujete neku kontrolu (npr. labelu) koju želite lokalizovati
- S desne strane u panelu Properties imate polje Text. Kada postavite kursor miša desno od tog polja pojaviće se mali točkić. Kliknite na njega i izaberite opciju **Replace with internationalized String**
- Labela će postati **%key.unspecified**. Umjesto toga možete staviti neku ključnu riječ koja označava jedinstveno riječ "Autor" npr. **%autor** (procenat se automatski ubacuje). Kasnije će svugdje %autor biti zamijenjeno sa odgovarajućom riječi u željenom jeziku





# Lokalizacija korisničkog interfejsa

- Ako pogledate sada u FXMLu vidjećete tagove oblika:  
`<Label text="%autor" />`
- Sada možete otvoriti u resources vaš Resource Bundle 'Translation'. Vidimo da u njemu imamo fajlove sa ekstenzijom properties npr. Translation\_bs.properties
- Tu je i fajl Translation.properties koji sadrži default jezik (ako aplikacija ne može odrediti korišteni jezik)
- U ovom fajlu pišemo niz poruka oblika:  
    autor = Author  
    naslov = Title  
itd. dakle ključna riječ koju ste odabrali ranije (bez znaka %, tj. znak =) i prevod na željeni jezik

# Pristup prevodu iz koda

- Ako u kodu negdje želite očitati string iz ResourceBundle koristite jednostavno metodu **bundle.getString("ključ")**  
`primaryStage.setTitle(bundle.getString("appname"))`  
;
- Prilikom pokretanja biće detektovane regionalne postavke vašeg operativnog sistema i automatski će biti odabran određeni jezik
- Možete nametnuti jezik sa `Locale.setDefault`:  
`Locale.setDefault(new Locale("bs", "BA"));`
- Primjer: [P22\\_BibliotekaLocalized.zip](#)

